AFRL-IF-RS-TR-1998-13
Final Technical Report
March 1998

# EXPERIENCE WITH ADAPTIVE SECURITY POLICIES

Secure Computing Corporation

Michael Carney, Brian Loe, and Terrence Mitchem

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**
**ROME RESEARCH SITE**
**ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1998-13 has been reviewed and is approved for publication.

APPROVED: Emilie J. Siarkiewicz

EMILIE J. SIARKIEWICZ
Project Engineer

FOR THE DIRECTOR:

WARREN H. DEBANY JR., Technical Advisor
Information Grid Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFGB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

## REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*

**1. AGENCY USE ONLY** *(Leave blank)*

**2. REPORT DATE**
March 1998

**3. REPORT TYPE AND DATES COVERED**
Final Jul 96 - Jan 98

**4. TITLE AND SUBTITLE**
EXPERIENCE WITH ADAPTIVE SECURITY POLICIES

**5. FUNDING NUMBERS**
C - F30602-96-C-0210
PE - N/A
PR - 1069
TA - 01
WU - P5

**6. AUTHOR(S)**
Michael Carney, Brian Loe, and Terrence Mitchem

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Secure Computing Corporation
2675 Long Lake Road
Roseville MN 55113

**8. PERFORMING ORGANIZATION REPORT NUMBER**
N/A

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Air Force Research Laboratory/IFGB
525 Brooks Road
Rome NY 13441-4505

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
AFRL-IF-RS-TR-1998-13

**11. SUPPLEMENTARY NOTES**
Air Force Research Laboratory Project Engineer: Emilie J. Siarkiewicz/IFGB/(315) 330-2135

**12a. DISTRIBUTION AVAILABILITY STATEMENT**
Approved for public release; distribution unlimited

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

This final report contains results from studying several aspects of adaptive security policies, including an assessment of the impact on assurance evidence from the loss of tranquility assumptions, the usefulness of audit during recovery from relaxed security, the use of a tool for specifying security databases, and trade-offs associated with different mechanisms for implementing adaptive security.

**14. SUBJECT TERMS**
Computer Security, Adaptive Security Policies, Dynamic Security Lattices, Policy Enforcement Separation, Security Database Specification, Formal Assurance

**15. NUMBER OF PAGES**
76

**16. PRICE CODE**

**17. SECURITY CLASSIFICATION OF REPORT**
UNCLASSIFIED

**18. SECURITY CLASSIFICATION OF THIS PAGE**
UNCLASSIFIED

**19. SECURITY CLASSIFICATION OF ABSTRACT**
UNCLASSIFIED

**20. LIMITATION OF ABSTRACT**
UL

# Contents

# List of Figures

# List of Tables

Section 1
# Introduction

## 1.1 Identification

This document is the final technical report for the project *Experience with Adaptive Security Policies* completed under contract number F30602-96-0210 for Rome Laboratory. The objective of this work was to extend the work done on the project *Experimentation with Adaptive Security Policies* under contract F30602-95-C-0047. In particular, the scope of the current project was to assess the following four items:

- the impact on system assurance of switching between policies,
- the usefulness of auditing in the switching/recovery process,
- tools to facilitate construction of security databases, and
- trade-offs regarding mechanisms for adapting policies.

This document is structured to present the results, conclusions and lessons learned from these evaluations. Experimentation with adaptive security policies for the current work was performed using the Distributed Trusted Operating System (DTOS). Since the work on these four tasks proceeded independently, the four sections of this document dealing with each task may also be read independently.

## 1.2 Experience with Adaptive Security Policies

As in [22], the appropriate place to begin is with the Orange Book definition of a security policy: *the set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information.* [17] When an organization employs an automated information system (AIS), the security policy for the AIS is an extension of the organization's security policy. Since many organizations work in a changing security environment, an organization's security policy, and hence the security policy that applies to any AIS deployed by that organization, must be adaptive. The goal of this work is to gain additional insight into the details of implementing and assuring an AIS with an adaptive security policy.

Section 2 describes the impact that the loss of tranquility assumptions has on formal assurance for an AIS with an adaptive security policy. Typically, static security policies are built upon a set of tranquility assumptions; it is assumed or explicitly stated that the security attributes of system entities (e.g. sensitivity level), or the rule sets that define the access rules between those entities (e.g. the level POSet), do not change. Since the very goal of an adaptive security policy objective is to change the nature of the access rules enforced by the system, it is a necessary conclusion that some tranquility assumptions will not hold through a policy transition. Section 2 lists a number of scenarios in which security policies might need to be adaptive. It also provides a description of a number of security mechanisms, tranquility assumptions that might be made in policies for systems enforcing those mechanisms, and a mapping back from the list of tranquility assumptions to the scenarios in which these assumptions might not hold. One topic of particular interest is dynamic lattices: adaptive security policies enforcing multi-level security (MLS) rules in which one does not assume that the lattice (level POSet) is tranquil across policy transitions. Section 2 proposes the use of a concrete representation of the lattice structure which facilitates the insertion of new nodes into a lattice or for collapsing a lattice. Along with dynamic lattices, a High-Water Mark Confidentiality Audit Policy is defined which would aid in analyzing information flow following a period of relaxed MLS rules (a collapsed lattice).

In Section 2.6 there is a description of the changes required to assure a system with an adaptive security policy as compared to a system with a static policy. Formal assurance tasks include security policy modeling, writing specifications, proving the model and the specifications are consistent, conducting analyses for covert channels, and verifying the correspondence between the specifications and code. Section 2.6 describes the additional burden for assurance tasks at a general level for any adaptive policy and at a detailed level for the DTOS medical demonstration [23] which was modified to include a second security policy for the sake of this discussion.

Audit data is often cited as a useful source of information about events which occur during periods of relaxed security. Section 3 presents the results of a study of the usefulness of audit data for system administrators recovering from a period of relaxed security. One of the main problems with audit data is that it is collected at such a fine level of granularity, usually for individual permission checks, that little useful information about the high-level flow of information can be extracted from it. One solution proposed in Section 3, and implemented for this contract, builds on the existing inter-process communication (IPC) protocols by applying a tracing identifier (TID) to each message passed through the system. The TID can be appended to audit data and used to collect individual audit events according to the thread of execution that has initiated them. Another common problem with audit data for client-server architectures is that many of the security-critical operations are performed by servers other than the microkernel. Since altering each server to audit events would complicate the integration of new servers, a modification to the microkernel was implemented to allow the microkernel to audit the requests made of other servers. Both methods for enhancing audit data were evaluated for their ability to aid a site recovering from a period of relaxed security.

Any AIS which is expected to enforce a security policy must initialize the security policy from a database that it can read and interpret. Section 4 considers the use of tools for specifying security databases for adaptive security policies. In particular, with an adaptive security policy, there are two or more policies under which the AIS may run, and therefore it is necessary to construct two or more databases to define the security policy after each transition in addition to the initial definition. It is already a difficult task to relate the policy specification to the organizational policy, and maintaining a large database using only a text editor is prone to error. With an adaptive policy, the problem is complicated by the necessity of managing the information that must change from one policy to the next. Section 4 explores using a tool with a graphical user interface (GUI) to specify the security database, describing the design and implementation of a GUI tool that creates security databases and supports the use of policy adaptation mechanisms.

The final section, Section 5, compares four methods for implementing adaptive security policies. The implementations are evaluated against a set of six criteria. One criterion is the range and type of policy transitions that the mechanism will support; this is referred to as policy flexibility. The second criterion is the functional flexibility, the effect that a policy transition has on running applications and the consequences for the users and their tasks. The remaining criteria include the security, assurability, reliability, and performance. The intent of the trade-off study is not to recommend a single solution for all implementations of adaptive security policies, but rather to show which mechanisms are most appropriate for a given application.

Although the four major sections of this report are somewhat independent of one another, the ultimate goal of the entire work is to take adaptive security out of the realm of theory and into the realm of application. Thus, in a more theoretical work, one might state that additional auditing would be useful in an AIS with an adaptive policy, but this study has attempted to analyze specific improvements to auditing in an existing system. To the extent that the development of a complete adaptive system was not within the scope, the observations of this report remain somewhat speculative; however, it is hoped that such observations lead to more concrete work along the path to developing adaptive secure systems.

## 1.3 Document Overview

The report is structured as follows:

- Section 1, **Introduction**, provides an overview of the document.
- Section 2, **Tranquility Study**, studies the sensitivity of assurance arguments to the loss of tranquility assumptions.
- Section 3, **Audit**, investigates the usefulness of audit data for recovery from periods of relaxed security.
- Section 4, **Database Tools**, explores the use of a GUI tool for specifying the security databases for an AIS operating with an adaptive security policy.
- Section 5, **Trade-off Study**, compares four different implementations of adaptive security using six criteria to evaluate which implementations are suitable for specific scenarios requiring adaptive security policies.
- Section 6, **Summary**, includes a summary of the results and observations of the project along with "lessons learned" and suggestions for future work.
- Appendix A, **DTOS Overview**, covers the background necessary to understand elements of this report specifically referring to the DTOS security architecture.

*Section* 2

# Tranquility Study

## 2.1 Introduction

The goal of this study is to understand the sensitivity of assurance evidence to the loss of tranquility assumptions.

Assurance evidence provides confidence that one knows precisely what happens, with respect to security, in a computer system, and that what happens falls within the bounds of a certain set of rules and requirements. If one cannot assume tranquility of the attributes of processes and files on a given system, or if the rules for access control are changing, can one still have confidence about the security of the system? How can one maintain a level of confidence in an AIS if common tranquility assumptions are removed? Those are the question which this study intends to analyze.

## 2.2 Overview of the Tranquility Study

In the first few subsections, in Section 2.3, this study takes a step back to survey the fundamental issues in assuring a computer system, starting with a statement of general security objectives for computer systems whether they employ adaptive policies or not. This is followed by a discussion regarding the refinement of objectives into concrete requirements which define who the authorized users are and what the authorized actions are relative to an organization's security needs and the mechanisms of a system. Since the objectives of a system must match those of the organization, the final subsection of the introductory material, Section 2.3.4, contains a listing of scenarios which require adaptive security policies. The scenarios are divided into six general categories: release and dissemination, roles and tasks, selective hardening, organizational support, and change of operational control.

The next several subsections in Section 2.4 contain a discussion of tranquility issues, giving a definition of tranquility and stating the utility of tranquility assumptions. Some of the common tranquility assumptions are explored. These include assumptions for systems enforcing multilevel security (MLS) controls, type and domain (or role-based) controls, and identity-based access controls (IBAC).

Some effort was expended to map the list of common tranquility assumptions in Section 2.4 back to the scenarios listed in Section 2.3.4, but many experienced readers may simply choose to skim or skip these sections since they contain background material familiar to many people in the computer security community. New ground is broken in the following sections on changing and enforcing an adaptive policy and in the section on the ramifications of non-tranquility for assurance tasks.

In the next subsection, Section 2.5, this report focuses on changing and enforcing an adaptive policy, considering dynamic lattices and presenting a concrete approach to defining a security level POSet which allows for insertion of dynamic levels (and for collapsing levels) as required under some scenarios. Moreover, this study proposes a high-water mark confidentiality audit policy in Section 2.5.3 which could accompany the use of dynamic lattices. This type of confidentiality policy corresponds to a low-water mark integrity audit policy in the way that the

Biba integrity model correspondd to the Bell and LaPadula confidentiality model. This particular policy defines a method for tracing contamination during times when the policy is relaxed, giving administrators useful information on how to roll-back to more restrictive policies.

Finally, in Section 2.6 this report presents the ramifications of non-tranquility for specific assurance tasks, including the following list: policy modeling, formal specification, proofs and arguments showing that the specifications are consistent with the policy model, and covert channel analysis. The general discussion is specialized to a specific case patterned on the DTOS medical demonstration in Section 2.6.5.

## 2.3 Security Objectives & Policies

### 2.3.1 Security Objectives

When setting out to write a security policy for an automated information system (AIS), whether the policy should be adaptive or not, one must decide on the objectives of the policy. As stated in [6] the most general objectives for an automated information system are for confidentiality,[1] integrity, and availability.

> **The Confidentiality Objective** Information is protected from improper disclosure.
>
> **The Integrity Objective** Data has at all times a proper physical representation, is a proper semantic representation of information, and is operated upon correctly by authorized users and information processing resources.
>
> **The Availability Objective** Information and information processing resources both remain readily accessible to their authorized users.

One might also choose to include an objective for accountability as well. An accountability objective might require that all events in which sensitive information is observed or critical information is altered shall be subject to auditing and that audit records shall be tamper-proof. The Orange Book [5] places equal emphasis on the accountability of the users of the system and on the mandatory and discretionary access controls. While accountability is an important feature of a system with an adaptive security policy, especially when rolling back permissions from a period of relaxed security, in this portion of the report, we shall concentrate on the aspects of assurance which are separate from auditing.

### 2.3.2 Security Policy Requirements

A security policy is a refinement of a set of security objectives into a set of security requirements. Since security objectives will be met in part by a set of mechanisms on the system, security policy requirements must be written to utilize those mechanisms. There must also be an argument which shows that the security policy requirements taken as a group are sufficient to meet the security objectives.

The confidentiality objective can be met by imposing access controls such as MLS, Type Enforcement [1], and IBAC. These satisfy the objectives of confidentiality and integrity. Access controls address confidentiality by constraining who observes data, and this is sufficient unless one also is concerned about covert channels. Access control is also sufficient to control who

[1] In some sources the word *security* is used to imply that information is not disclosed improperly. This sense of the word is too narrow, because it does not include the concept of protecting resources from being interfered with. If one intends to prevent unauthorized disclosure of information, then confidentiality describes this more accurately.

modifies data on the system, but is insufficient to insure that the integrity of information is maintained. For example, Clark & Wilson [7] refer to business transactions as being well-formed transactions, e.g. ledger entries in a double-entry bookkeeping system have to keep the books in balance. Access control only insures that the integrity of system information is maintained (or lost) through the actions of authorized users.

The objectives stated thus far are very broad, general statements about security objectives and will apply to many situations and organizational security objectives. Thus, to be useful a policy needs to be refined in a step-wise fashion so that the policies of the organization deploying a trusted system are reflected in the security policy that the system is designed to uphold. The step-wise refinement depends on the definition of who is authorized to observe and modify data, and the set of mechanisms used to enforce them.

For example, in an environment in the national security establishment, files have sensitivity labels and users are cleared according to the trust afforded them. Therefore, a system deployed in this environment would be expected to apply MLS rules to access control.

This suggests that in practice a security policy is not necessarily formulated in a top-down method, but in a combination of top-down and bottom-up steps in which the organizational or system-level security objectives are refined into lower-level requirements in the AIS with the mechanisms for enforcement in mind. Therefore, for a system enforcing MLS rules, the confidentiality policy can be refined into a requirement that no one shall observe data unless authorized to do so by MLS rules, i.e., the Bell and Lapadula's simple security rule.

### 2.3.3 Adaptive Security Policies

Since the security policy enforced by an AIS must reflect that of the organization deploying the system, the security policy must be adaptive in the sense that the set of individuals who are authorized to observe or alter information may be time dependent. The basic objectives of the security policy have not changed, but the rules by which the objectives will be met have, and the method for refining the objectives into concrete requirements has changed as well.

The question is to what extent are you able to meet your objectives when the rules defining authorization change. An immediate change of policy suggests that if individual A can access item B under one scenario, and the rules change so that A is not authorized to access B, the system should no longer allow A to access B.

However, there may be circumstances under which we would like to allow A to have access to B, even though they are officially no longer authorized to do so. For example there may be some critical task which A must be allowed to complete even with the change of policy. If the system does allow A access to B after the rule change, there should be a period of time after which access is denied. In such situations there must be a transitional policy which allows continued access.

### 2.3.4 Scenarios Requiring Adaptive Security Policies

The following subsections and itemizations were extracted from electronic correspondence between the Institute for Defense Analysis and Secure Computing [15] regarding adaptive security policies in the "real world."

2.3.4.1 Release and Dissemination The release and dissemination of sensitive material is often time dependent. There are a number of similar scenarios in which the authority to observe a file is time dependent.

- It may be necessary for a file, which a party is not allowed to observe today, to be released to that same party by a specific time tomorrow.
- Alliances can change so that allies with whom one currently shares a particular form of data (e.g., image data) today might not be allowed to see the next version of the data.
- Different rules may apply for operational plans between the time when one is planning the operation and executing it. The "need to know" rules change between the two stages of planning and execution.

Most of these scenarios depend on discretionary access control enforced through IBAC policies and access control lists. An adaptive policy meeting the needs of an organization under the preceding scenarios would need to have time dependent access control lists. In the case of operational plans to which access is more highly restricted during the planning phase, there may be a mode change from peacetime to crisis which triggers the change in the ACL making the plans more widely disseminated for those who need to execute the plans.

When a document is subject to timed release, it is necessary for the release time to be adjustable. Policies for systems which employ timed release must account for when and how are such releases authorized, authenticated, and executed. Both the authorization and execution may be separate operations each of which may be time dependent.

2.3.4.2 Roles/Tasks/Domains It is a common scenario that two or more people will be responsible for carrying out the duties of one role, but not concurrently (e.g. Watch Officer). In these cases it is necessary for one user to hand off responsibilities of the role to another user. If the user currently acting in the role is forced to log out before his successor is allowed to log in, tranquility assumptions may still hold.

However, there be cases in which the function of the role need to be on-going, without interruption. In such a case there should be a mechanism for transferring control of the subjects operating on behalf of the current user to his successor. Possibly an additional subject would need to be invoked in order to authenticate the identity of the successor, and accomplish the transfer of authority with full accountability for the users involved.

A complicating factor to consider is when the two users involved in a transfer of role are also authorized to other roles. For example, a Watch Officer may also be the Chief Engineer. As Chief Engineer he may need to have access to information which is not necessarily available to other users authorized to just the Watch Officer role. If a user is authorized to multiple roles, the policy must account for whether the user is allowed to operate in several roles simultaneously or whether certain tasks authorized to some roles are so sensitive that the user operating in that role is not allowed to be acting in any other role simultaneously. This would comprise the notion of "separation of duty."

Similarly, a user acting in one role may be limited in his actions during normal operations, but under exceptional circumstances he may need additional authority usually reserved for a user operating in another role. For example the Command Duty Officer may need to act with the authority of the Commanding Officer. This authority, while delegated to the Command Duty Officer by the Commanding Officer, can only be invoked under specific circumstances and any transition in the system which allows the Command Duty Officer to invoke these changes must be highly regulated and audited to insure accountability. Some of the caveats in the preceding paragraph apply here when users can authorize themselves to roles with greater privileges.

2.3.4.3 Selective Hardening A number of papers on adaptive policies have assumed that in crisis situations that security constraints may be relaxed because more users have a "need to know" in times of crisis or because the cost of losing the system exceeds the cost of a breach of security. However, there are conditions under which security configurations may need to be "hardened" as defensive conditions change, e.g., based on a "DEFCON" alert or an anomalous event.

Under this scenario, a policy may be hardened by reducing access through the usual access control policies such as mode dependent access control lists or a more stringent type enforcement policy. A policy could also be hardened by using a larger level POSet during the period of selective hardening and by using a collapsed security lattice during normal operation.

A policy could also be hardened through measures other than the usual access control mechanisms such as full vs. selective auditing, stronger cryptography, prioritized resource sharing, and enabling of redundant resources and protection features.

2.3.4.4 Organizational Support A task force is typically composed of units from several organizations. The units comprising the task force must come together in a time constrained manner, and the composition of the task force may change when either some units discontinue participation or others join the task force. Under this scenario, the management of the security policy becomes an issue in a domain where there may be little time to devote to security concerns. Only a policy which is well planned in advance of operations can support such dynamic changes.

2.3.4.5 Change of Operational Control (CHOP) A unit under the control of one command may be reassigned to support a mission under the command of another organizational unit. The security policy of that unit must flexible enough to be reassigned to the new command in a seemless fashion. For example, an AWACS plane flying over Bosnia, may be suddenly tasked to support a NATO maritime interdiction mission with the Italian and U.S. Navies in the Adriatic. The security policy for the AWACS systems must accommodate the change of operational control. Some information from the AWACS plane would necessarily be shared with the new command. Currently, there are no fully automated forms for this type of policy.

## 2.4 Tranquility Issues

### 2.4.1 Definition of tranquility

What do we mean by tranquil? Webster's Dictionary defines[2] it to be "unvarying in aspect," and offers "steady" and "stable" as synonyms. For the sake of studying security policies regarding the confidentiality and integrity of data, we define a relationship between entities, say *A* and *B*, on the system to be tranquil if the access control policies state that the accesses permitted for *A* to *B* do not change.

### 2.4.2 Utility of Tranquility Assumptions

Tranquility is assumed in a security policy for various attributes of entities or for the relationships between attributes for the sake of keeping the security policy both strong and simple.

[2] The second definition from Webster's Ninth Collegiate Dictionary.

To state tranquility of an attribute such as the sensitivity label as a system requirement is a strong requirement. For example, for MLS mandatory access controls, Part II, Section 5 of the Orange Book [5], states that "the system must assure that the designations associated with sensitive data cannot be arbitrarily changed, since this could permit individuals who lack the appropriate authorization to access sensitive information." A system that requires tranquility of sensitivity labels assures *a fortiori* that this particular attribute cannot be "arbitrarily changed" since they cannot be changed at all. Furthermore, tranquility assumptions make aspects of the policy simpler, because the AIS intended to enforce the policy is easier to implement and analyze, and hence is more likely to meet the requirements of the security policy. Thus, tranquility assumptions allow one to state some security requirements more strongly and ease assurance tasks.

The problem with tranquility assumptions are that they are inflexible. Furthermore, they do not map well to many common scenarios which require greater flexibility. As stated above, it is also important that the security policy enforced by an AIS model the security policies of the organization.

### 2.4.3 Common Tranquility Assumptions

Where do we ordinarily assume *tranquility*? Basically, entities have attributes, and accesses between entities are allowed or denied based on those attributes and a set of rules for access control. The security policy is defined on the system by the combination of attribute assignments (labeling of entities) and access control rules. We can consider three types of access control: MLS, Type Enforcement, and IBAC. The former two are mandatory access control mechanisms, and the last one is a discretionary access control mechanism. How these policy permissions are enforced is another matter.

The entities on a system consists of subjects (programs in execution), objects (data storage containers), and users.[3] The attributes and rule sets required for enforcing each type of policy are summarized in Table 2-1.

Table 2-1: Attributes and Rule Sets for Enforcing Security Policies

| | MLS Policies | Type Enforced Policies | IBAC Policies |
|---|---|---|---|
| Subject | Level | Domain | User & Group |
| Object | Level | Type | Access Control List |
| User | Level(s) | Role(s) | Group(s) |
| Rule Definition | Level POSet | Type Enforcement Database | Access Control List or UNIX-like Protection Mechanisms |

[3] Users are system entities that internally represent individuals who use the system. Attributes assigned to users are assumed to correspond to authorizations granted to individuals. Devices could be considered to be separate entities, but for the sake of simplicity, devices will be assumed to have attributes similar to objects. For example, an access control list for a device would define the set of users and groups who may access a device.

2.4.3.1 Tranquility and MLS Policies Systems used in national security typically employ hierarchical levels and categories for defining the sensitivity of data and the clearance of individuals to access that data. The traditional rules for Multilevel Security (MLS) are implemented to prevent users with low clearances from observing high-level data and to prevent high-level data from being passed to lower-level containers where users without adequate clearance may see it.

Since subjects operate on behalf of users, it is necessary to assign levels to subjects as well as objects. The typical rules for enforcing an MLS policy at the level of subjects and objects are the Bell and LaPadula simple security and *-properties. However, it is generally necessary to relax these rules somewhat to allow certain subjects to write information down in level (downgrade). Trusted permissions such as this may be granted through a number of means either through a relatively flexible mechanism such as a type enforcement or less so by hardcoding permissions for subjects which run with fixed identifiers. Most systems have a notion of downgrade privilege since the strict adherence to the simple security and *properties limit the utility of the system.

For an MLS system, typical tranquility assumptions include that the level assigned to each subject and object is tranquil for the duration of that entity's existence. Similarly, one might assume that the sets (or classes) of privileged subjects which are granted permission to downgrade are tranquil.

For a non-tranquil MLS system, we may allow the level of an object to be changed. This would be a violation of the strict Bell and LaPadula rules forbidding no write down. However, sets of permissions in a tranquil system may be logically equivalent. For example, a high-level subject may have permission to downgrade information from one object to another if it has the following three permissions: premission to create a low-level object, permission to read data from a high-level object, and permission to write that data to the low-level object. The task of assuring this operation is probably equivalent to assuring the operation of allowing a trusted subject to change the sensitivity label of the object. This may have other minor benefits, say for the performance of the system, but these are necessarily secondary to the strength of the assurance of the operation.

Similarly, for a non-tranquil MLS system, we may allow the level of an subject to change. If a subject drops in level, any data it holds in local memory would effectively be downgraded. Furthermore, if a high-level subject can drop in level, then to enforce the *-property, permissions to read high-level data which the subject had been granted prior to its change in level must be invalidated in any permission caching mechanisms. Conversely, if a low-level subject can be raised in level, permissions to write data to low-level objects which the subject had been granted prior to its change in level must be invalidated.

Flushing caches of permissions might be handled in different ways, these are discussed in [22]. However, it must be noted that there are tradeoffs depending on whether permissions are recomputed as needed or all at once. Furthermore, one may wish to restrict a subjects ability to change in level depending on which objects it has open. It may be necessary for a subject to close certain objects before changing level.

Systems in which the level of a subject is not tranquil may assign a set of allowable levels to that subject. Such sets could be represented by a maximum allowable level, a minimum allowable level, or a range with both a maximum and minimum level.

User levels are usually treated somewhat differently than subject levels. A user will typically have a set of levels associated with it representing the set of levels at which the user is allowed to operate. Some systems also record the current level at which a logged in user is operating, and the user may change the level at which he operates as long as it is within the set of allowable levels. Otherwise the user may have subjects operating on his behalf as long as the subjects operate at a level which is within the allowed range.

Whether a specific level is associated with a user at any given time, or if the user is allowed to have subjects operating at a range of levels, the question relative to MLS rules is whether the user can cause information to flow downward in level either through his own actions or through subjects at his disposal. Generally speaking, we have to assume that users who are cleared to observe certain levels of data can be trusted not to disclose that data. So even if a user can have two subjects operating at differents levels, information does not flow downward without passing through the user unless it is already allowed through existing exceptions to the Bell and LaPadula rules.

The remaining tranquility assumption in an MLS policy is that the mechanism (database) that define the access permissions allowed between entities, namely the level POSet, is tranquil. More specifically, the set of levels and categories is fixed, and the dominates relation which defines the partial ordering is also fixed. If levels are represented as a lattice, this means that nodes and edges on the lattice are not removed or added, and no set of nodes can be collapsed into a single node. Note that changing the level POSet by changing the dominates relation effectively changes the level of objects. Examples in which the level POSet fails to be tranquil are given below.

During a period of relaxed security (crisis), some levels could be collapsed together, e.g., SECRET and SENSITIVE (see Section 2.5.3). During a crisis, a user may be required to read a file normally unavailable to him; the insertion of new levels for the file and user may be required (see Section 2.5.2 and [13]). Collapsing levels is logically complementary to inserting levels. Instead of inserting new levels and relabeling a file, from $l_f$ to $l'_f$, and granting a higher clearance, from $l_u$ to $l'_u$, to a user who needs to read that file during a crisis, those labels and clearances could be pre-loaded. During normal operations, those pre-defined levels could be collapsed: $l'_u$ with $l_u$ and $l'_f$ with $l_f$.

One question remains when considering changes to the level POSet and that is how levels represented on an AIS map to levels defined in the world external to the machine. This, of course, is highly dependent upon the policy which the AIS is intended to implement.

#### 2.4.3.2 Tranquility and Type Enforced Policies

At the most basic level Type Enforcement simply assigns another attribute to each subject and object on a system. Each type of access that a subject requests for a given object is allowed or denied based on a comparison of the domain and type, and each type of access that a subject requests for another subject is allowed or denied based on a comparison of the domains for the requesting subject and the target subject. That is for every ordered pair of domain and type, there is a set of permissions (e.g., a subset of {read, write, execute, create, destroy}) which are the accesses allowed for a subject operating in that domain to any object of that type. Thus, all subject-object and subject-subject accesses are limited to a set consistent with the tasks which the subjects are designed to perform. This type of task-based access control mechanism is perfectly suited to enforcing least privilege.

Furthermore, since executable code is assigned a type and typically there is a one-to-one correspondence between domains and executable code, only one specific type of process may operate within a given domain. This implies that Type Enforcement provides a mechanism which not only restricts which subjects have access to objects but a mechanism by which one assures that subjects perform only those transformations of data which are intended.

Each user is assigned a set of domains in which he is allowed to have subjects operating. Since these domains define a set of tasks that the user can perform, a set of domains is referred to as a *role* for the user. Thus at the user-level, type enforcement is a role-based access control mechanism.

For a type enforced system, typical tranquility assumptions include the assumptions that the domain of a subject is tranquil, that the type of an object is tranquil, and that the type enforcement databases will not change.

As for roles assigned to a user, there might be tranquility assumptions at several levels. The most restrictive assumption would hold that the set of roles assigned to a user is tranquil. However, just as for the clearance levels assigned to a user, the roles of a user might be changed by a system administrator. We could characterize this as "tranquil except for system administration."

For a non-tranquil system, we may allow either the type or domain of an object or subject to change, we may enforce a different type enforcement table, or we might allow a user to change roles during an active session. Whereas in the MLS case loss of tranquility assumptions have clear implications relative to the Bell and LaPadula rules, the goals of type enforcement are to enforce least privilege and separation of duty. These goals are less concrete than the confidentiality goals represented by an MLS policy.

However, since type enforcement policies are a task-based policy at the level of subjects and objects, reasons for allowing non-tranquility are clear. An object's type could be changed if the data in the object were needed to perform tasks which have been assigned to subjects in distinct domains, or if the data needed to be processed in a number of steps by separate subjects in a "trusted pipeline." A subject's domain could be changed because it needs to perform a sequence of tasks as a "trusted procedure." The goal in this case would be to grant a minimal set of permissions to the subject for each task in the sequence.

The former example can be handled in a system with tranquil types where one subject can read an object of one type and write that information into an object of another (this is analogous to the effective downgrade mentioned in the previous subsection). The latter cannot be accomplished in a system with tranquil domains and potentially could be very useful especially when a subject is required to perform an action which is very sensitive and requires a change to a domain which might be highly privileged. Again, for non-tranquil type-enforced systems, as for non-tranquil MLS policies, permission caches must be flushed to prevent information from flowing through a subject which has changed domains.

Just as the POSet may change for a non-tranquil MLS policy, the type enforcement databases which contain the access privileges between domains and types or between two domains may change. For example a domain may be given a larger (or smaller) set of types to access in order to accomplish a wider (more constrained) set of tasks. Similarly, a domain may be granted more permissions to a type to which it was already granted access. For example, during a period of crisis, downgrade privileges could be given to a domain in addition to normal write permission. Opening up a type enforcement database by adding downgrades could be an alternative to collapsing levels which allows a finer granularity of control over how information can be disclosed during periods of relaxed security. As with previous examples, permission caches would need to be flushed when permissions have been revoked due to a policy change.

On LOCK [18], a role consists of a list of domains in which any user who is authorized to that role may have subjects operating. While there is no table listing each role and its set of domains on LOCK, the concept of a role table which maps a role name to a set of domains is a convenient construct. The purpose of imposing roles on users is to provide separation of duty and least privilege at the level of the system users rather than at the level of subjects or domains.

It is probably not reasonable to assume that a user cannot change roles or that the set of roles to which a user is authorized in tranquil. However, it might be useful to assume, for the sake of separation of duty, that a user cannot change roles unless all subjects associated with tasks in one role are terminated. This is a higher level of control required than for permission cache flushing for domain and level changes for subjects.

If the set of domains in a role changes, then the same problems occur as in the previous case in which permissions are added to or removed from domains, but it occurs again, as in the previous paragraph, at the level of users rather than subjects. It may be necessary for a user to terminate all subjects operating in domains to which he is no longer authorized.

#### 2.4.3.3 Tranquility and IBAC Policies

The tranquility assumptions for IBAC policies are somewhat different from those for MLS and type enforced policies. In particular, the rules by which IBAC policies are enforced are also the attributes associated with each object. Furthermore, IBAC policies have been generally applied to implement *discretionary* access controls rather than *mandatory* controls though there is no reason why they cannot be used for mandatory controls. That is, it is more likely to be the case that the Access Control Lists (ACLs) for objects are not assumed to be tranquil. Hence a measure of non-tranquility is already accepted for IBAC policies.

However, for IBAC policies a typical tranquility assumption holds that the user and group associated with a subject are tranquil. For IBAC policies which enforce mandatory access control, the ACLs associated with objects would be tranquil as well.

If a subject can be transferred from one user to another, supposing that the two users need act in the same role but need to hand-off responsibilities for that role, then the subject user would not be tranquil. However, we might assume that the two users belong to a group of users who would be fulfilling the obligations of this role, and group restrictions could be used to meet role restrictions. However, whenever a subject is handed off from one user to another, there may be problems with access control lists on the user-level. For example if user A hands off subject S to user B and A has discretionary access to a file via S but B is not on the access control list, then there may be problems in carrying out the hand-off. Either we need to ensure that S must close the file in question and purge its local memory of any data observed in the file, or in the case of write permission, the files must simply be closed before hand-off occurs.

Considering some of the scenarios discussed in Section 2.3.4, an adaptive security policy might employ non-tranquil ACLs which are either time or mode dependent. For example there may be one ACL for peacetime and one for wartime, or one for normal mode and one for crisis mode.

#### 2.4.3.4 Other Tranquility Assumptions

Other assumptions about tranquility occur at lower levels of the system such the Memory Management Unit (MMU). In this case, a subject can open an object and request permission to read, write, etc. On some systems, once the permission vector is computed and entered into the MMU, that access may not change. This can be changed on the Mach microkernel. In fact, since DTOS is based on Mach, the ASP 1 project made use of the DTOS facility for flushing cached accesses to this level of detail. This is true of systems which use a client-server architecture in which the server knows the contents and address of each file, and subject must make calls to the server to read and write data.

The concept of a Trusted Path, as represented in the Orange Book [5], represents a tranquility assumption of sorts: namely, that "the TCB shall support a trusted communication path between itself and users when a positive ...connection is required." It is conceivable that certain utilities which are designed to operate in trusted path mode in normal operation might not necessarily be allowed this uninterruptible connection during a crisis. Or in cases in which defenses must be hardened, and the integrity of input or output is considered more critical, utilities could be designated trusted path utilities under the hardened policy.

## 2.5 Changing & Enforcing the Security Policy

### 2.5.1 Changing Security Policies

In this section we list a number of different methods for changing the security policy and enforcing the new policy. In [22] there were three methods for changing security policies investigated. We repeat these here:

- Two or more policies may be incorporated into a single Security Server. The policy is changed by changing the definition of the permissions allowed by the system. This can be a matter of providing two different tables for computing access vectors.
- The current Security Server hands off the capability to receive access control requests to another Security Server which defines a different policy.
- An external agent designates a different port from which a different Security Server receives access control requests.

Logically, what occurs in each of these three methods is that the computation of access vectors changes; the attributes of the subjects and objects in the system have gone unchanged. Changes that can be implemented in this way include collapsing or inserting levels in an MLS system and opening up or closing down the Type Enforcement database to be less or more restrictive.

None of these three methods seems to be substantially different from the others when considering the end result: we have changed the policy definition. However, from an assurance standpoint the three methods of changing the policy are different because of the way that the transitions can be handled. What matters is how quickly these changes are implemented, the degree of atomicity of the changes, and how these changes are coordinated in the system across its components.

The first of these three is more nearly atomic, but it is also monolithic and inflexible. The second and third are multi-step processes: the Security Server must notify others that the policy is changing, transfer control to the new server, and the new server must receive authority from the old one. The Security Server must know that the policy has been transferred and, especially in the third method, that it has been transferred to the right server.

In order to compare the three methods, there are two questions that must be answered. First of all, from the point of view of assuring the system, how do you know with any confidence *what* policy is actually being enforced by the system? With an atomic change of policy and immediate cache flushing, uncertainties over the policy enforced during transition are decreased, if not nonexistent. However, with more complicated coordination between two servers or with delayed flushing of cached permissions, the boundaries between two policies being enforced are not cleanly delineated.

Secondly, there is the balance between functionality and security. Does the functionality allowed or precluded during the transition between policies meet the needs and objectives of the organization that the system is intended to serve? For example, if a user is in the middle of performing a task when the policy is made more stringent, the user, or any processes operating on his behalf, may no longer possess the permissions required to complete the task. Thus, a task may be unable to be completed if the new policy is enforced too quickly. Thus, functionality has been sacrificed for the sake of security. If completion of such tasks is mission critical, then the alternative, in which security is sacrificed for functionality, would be more desirable. Some of these trade-offs are described in greater depth in Section 5.

### 2.5.2 Dynamic Lattices

Dynamic lattices are described in [13] and [22]. The idea is that during a crisis a user at level $l_u$ may need to see a file at level $l_f$ even if $l_u$ does not dominate $l_f$. During a period of relaxed security, the file could be downgraded to $l'_f$ and the user could be allowed to operate at $l'_u$ where $l'_u$ does dominate $l'_f$. However, we do not wish to downgrade the file to a level which is observable by users who were previously unable to observe it (except for the one now operating at $l'_u$). Similarly, we do not wish to give the user now at level $l_u$ access to a whole set of files that he was previously unable to see. Thus for a level $l$ in the original lattice, $l$ is dominated by $l'_u$ if and only if $l$ is dominated by $l_u$. and $l$ dominates $l'_f$ if and only if $l$ dominates $l_f$.

Consider a typical lattice structure with two hierarchical levels, high and low ($H$ and $L$), and two compartments (or categories), $a$ and $b$, as represented in Figure 2-1.

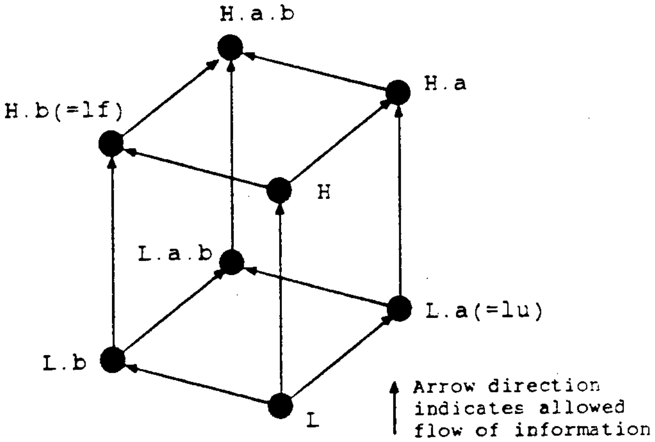


Figure 2-1: A typical lattice with two hierarchical levels and two compartments.

Suppose that $l_f = H.b$ and $l_u = L.a$. It's clear from the lattice in Figure 2-2 that it is not possible to insert new levels appropriately without adding more compartments even if you add an intermediate hierarchical level, $M$. The level $l'_u$ must include the compartment $a$. The level $l'_f$ must include no more than the compartment $b$. If $l'_f = M.b$, then $l'_u$ must be $M.a.b$ which dominates $L.a.b$ which is not dominated by $l_u = L.a$. If $l'_u = M.a$, then $l'_f$ must be $M$ which is dominated by $H$ which does not dominate $l_f = H.b$. Therefore, if this problem is to be solved, it must be done so by adding compartments as in Figure 2-3.

By adding the compartment $c$ to all levels which are not dominated by $l_u$, it is possible then, by not giving compartment $c$ to $l'_u$, to ensure that no level is visible to $l'_u$ unless it was also visible to $l_u$. Note that in Figure 2-3 that the intermediate level $M$ could in fact be equal to $H$.

How can this be done in general? By assigning a number to each node in the lattice, we can make the number play the role of $c$ as in the previous example. We number the eight nodes in Figure 2-1 from 1 to 8. We add a numerical compartment $n$, $1 \leq n \leq 8$, to a node $A$ if $A$ is not dominated by the node associated with $n$. By adding so many compartments, the choices for $l'_u$ and $l'_f$ become more numerous. Whether one can choose these in a unique way is unclear.

A better, more flexible way of creating dynamic lattices would be to use the POSet nature of the lattice structure and to worry less about hierarchical levels and compartments. The entire structure of a POSet can be localized to each node where it is only necessary to describe those nodes immediately above or below in the lattice. For example, at node $A$, one lists which

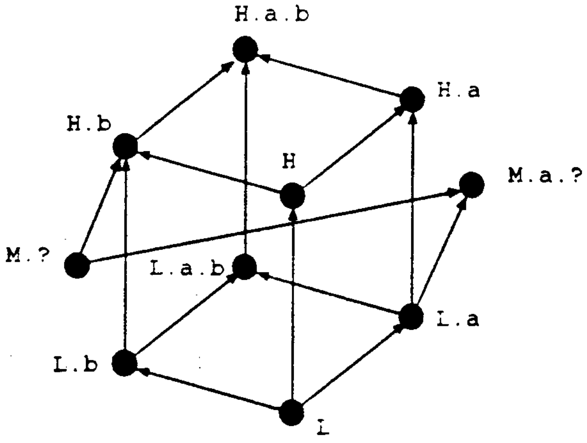
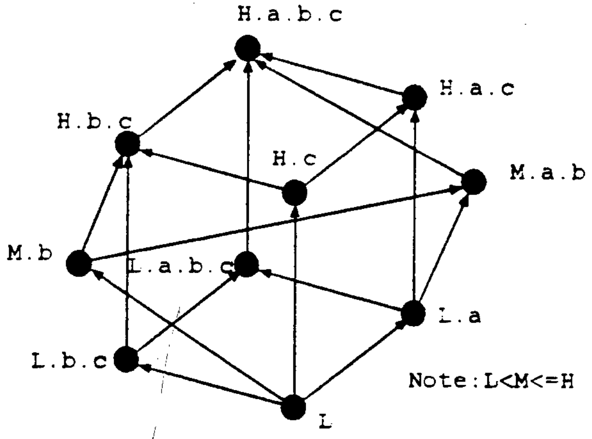


Figure 2-2: Inserting levels can cause problems.



Note:L<M<=H

Figure 2-3: Adding compartments makes the new lattice work.

nodes immediately dominate $A$ and which are immediately dominated by $A$. In Figure 2-1, $L.a$ dominates $L$ and is dominated by $L.a.b$ and $H.a$. Since the local description of the POSet tells us that $H.a.b$ dominates $H.a$ and $L.a.b$ as well, we have by transitivity that $H.a.b$ dominates $L.a$ as well. Note that by listing the levels above and below that the description is redundant and that either column alone will suffice to describe the lattice; however, it can be useful to record both sets of nodes which are directly above or below a given node.

Dynamic levels could be inserted by specifying new levels and their local dominance relations. Suppose once again that $l_f = H.b$ and $l_u = L.a$. The level $l'_u$ must dominate $l'_f$ and $l_u$. The level $l'_f$ must be dominated by $l'_u$ and $l_f$ (as seen in Figure 2-2). However, that only specifies half of the local description. What remains to be determined are the levels dominated by $l'_f$ and levels that dominate $l'_u$. The requirement is that any level that dominates $l_u$ and $l_f$ must still dominate $l'_u$, and that any level dominated by $l_u$ and $l_f$ must still be dominated by $l'_f$. Thus at $l'_u$ the levels above must include (be equal to) the least dominating level of $l_u$ and $l_f$ (in this case $H.a.b$), and at $l'_f$ the levels below must include the greatest dominated level of $l_u$ and $l_f$ (in this case $L$). In our example, Table 2-3 yields the same result as Figure 2-3.

Table 2-3, in fact, does not take full advantage of the local structure of the POSet. Without the

| Node | Dominated by | Dominates |
|---|---|---|
| H.a.b | — | H.a, H.b, L.a.b |
| H.a | H.a.b | H, L.a |
| H.b | H.a.b | H, L.b |
| H | H.a, H.b | L |
| L.a.b | H.a.b | L.a, L.b |
| L.a | H.a, L.a.b | L |
| L.b | H.b, L.a.b | L |
| L | H, L.a, L.b | — |

Table 2-2: Local POSet Structure of Figure 2-1

| **Node** | **Dominated by** | **Dominates** |
|---|---|---|
| $l'_u$ | H.a.b | L.a, $l'_f$ |
| $l'_f$ | H.b, $l'_u$ | L |

Table 2-3: Additional Structure for $l'_u$ and $l'_f$

restrictions placed on the new levels by compartments, the levels $l'_u$ and $l'_f$ could be one and the same as shown in Table 2-4.

| **Node** | **Dominated by** | **Dominates** |
|---|---|---|
| $l'_u\ (= l'_f)$ | H.b | L.a |

Table 2-4: Alternative Structure for $l'_u = l'_f$

If one is inserting levels, then one might want to maintain consistent local descriptions of levels above or below, but since the local description including both is redundant, it may be unnecessary to insert the new levels into the descriptions of levels above/below for the old levels as long as the mechanism enforcing the MLS rules "knows" how to read the level database and can account for the lack of redundancy.

Similarly, one might also collapse security levels using the local structure of the POSet. For example, if it was necessary to collapse the hierarchical levels $H$ and $L$ while maintaining the compartments $a$ and $b$, the local POSet structure could be amended so that any two nodes in Figure 2-1 which are being collapsed into one node each dominate the other. For example, $H.a$ dominates $L.a$ and is dominated by $L.a$. It would be unnecessary to specify all other relations which are true in the collapsed lattice (e.g., $L.a.b$ dominates $H.a$, since $L.a.b$ dominates $L.a$ and $L.a$ dominates $H.a$). This provides considerable flexibility in collapsing a POSet, though any organization collapsing levels would need to consider all of the ramifications of collapsing levels in this way.

Thus, the local structure of POSets provides a framework that is highly flexible. However, the lattices and the local POSet structure provide mathematical models for a relation on a set of external levels as well as internal levels, and the foremost consideration is to map any external levels into the internal levels for an AIS.

As an alternative to the two methods for representing dynamic lattices in a system listed above, one might use MLS and type enforcement to enforce an adaptive security policy in which a domain which is granted read and write privilege in the more restrictive database could be granted read and trusted write in the relaxed database. Thus, while a file could be downgraded under the relaxed policy, type enforcement provides a finer granularity of mandatory access control than MLS alone, and access to downgraded objects would still be subject to access controls based on task or role considerations. Downgraded files could not be observed by arbitrary users simply because they were cleared to view them.

### 2.5.3 A High-Water Mark Confidentiality Audit Policy

According to [6], there are six variations of the Biba Integrity Model: the Low-Water Mark Policy, the Low-Water Mark Policy for Objects, the Low-Water Mark Integrity Audit Policy, the Ring Policy, the Strict Integrity Policy, and the Discretionary Integrity Policy (a specific proposal for the Multics system). The first five of these are described in detail in [6]. The most commonly addressed variation of the six policies is the Strict Integrity Policy which is dual to the Bell and LaPadula model for confidentiality. This paper introduces a policy which might be useful for addressing concerns about confidentiality when recovering from a relaxation of security policies which is dual to the Low-Water Mark Integrity Audit Policy. This new model shall be referred to as the High-Water Mark Confidentiality Audit Policy.

Whereas the Strict Integrity Policy provides a measure of integrity for subjects and objects, the Low-Water Mark Integrity Audit Policy provides a measure of contamination. Whenever a subject observes an object at a lower contamination level, the contamination level of the subject is lowered to match that of the object, unless the contamination level of the subject was already less than or equal to the contamination level of the object. Similarly, when a subject modifies an object at a higher integrity level, the integrity level of the object is lowered to match that of the subject unless the contamination level of the object was already less than or equal to the contamination level of the subject.

For a High-Water Mark Confidentiality Audit Policy, instead of the assigning a new set of levels of contamination for subjects and objects, the policy would assign a second sensitivity label from the existing set of sensitivity levels used for enforcing MLS policies. Initially, subjects and objects would be assigned contamination levels equal to their sensitivity levels.

If security levels are collapsed during a period of policy relaxation, a subject which was originally at the secret level might read an object which was originally classified as top secret. Since the level POSet has been collapsed, the system would not deny the subject access to the object; however, as a result, the contamination level of the subject would be raised to top secret. Similarly, if a (formerly) top secret subject modifies a secret object, the contamination level of the secret object would be raised to top secret to match that of the subject.

Thus, after a period of security relaxation, there may be a number of objects and subjects which have contamination labels which do not correspond to their sensitivity labels. By determining which subjects and objects fall into this category, one can then assess the level of contamination incurred by lower-level objects and subjects while security was relaxed.

Beyond the possibility of merely auditing the information, when the period of relaxed security is over and it is necessary to roll-back to more strict controls, it might also be possible to reclassify subjects and objects to levels appropriate for their level of contamination. For example, any object which was at one time labeled secret, but was contaminated to the level of top secret in the interim, would be reclassified as a top secret object upon roll-back. Objects which have been raised in level at roll-back, but were not contaminated during the relaxed policy, could subsequently be reclassified to their original levels after examination by an authorized individual.

## 2.6 Ramification of Non-Tranquility for Assurance Tasks

In this section we offer some general comments on the impact of an adaptive security policy to specific assurance tasks. These tasks include the following:

- Policy Modeling
- Formal Specification
- Constructing Proofs that the Policy is Satisfied
- Spec-to-Code Analysis
- Covert Channel Analysis

While some of the information in this section appears in earlier sections of this report, we repeat it here to emphasize the impact on the assurance of systems which do not assume tranquility.

### 2.6.1 Policy modeling

As stated in Section 2.4.2, the value of stating security policy requirements which require tranquility assumptions is that the policy is both simple and strong.

If the security policy will be formalized in a specification language such as *Z* [25] or *PVS* [19], requirements which explicitly state the tranquility assumptions become very powerful tools. Tranquility assumptions form system invariants. Knowing that these invariants exist make the task of writing a consistent set of security requirements easier.

However, the restrictive nature of tranquility assumptions limits the range of expressiveness for security policies. This limits the utility of writing a security policy, since many organizational policies are not tranquil, and the policy written for the AIS must map to the policy applied to the organization using the system.

First and foremost, an adaptive security policy has to address how the policy itself changes over time. For attributes which are allowed to change, it may be necessary to state under which conditions they may change. There may be policy statements about the changing of policy rules and who or what may effect these changes. While statements in which tranquility assumptions are explicitly stated may be eliminated from an adaptive policy, it is not clear that the number of statements would decrease.

Furthermore, an adaptive policy may require a number of changes which make the policy inherently more difficult to write. In particular, one might need temporal or real-time logic to state various policy requirements. For example, a policy requirement might state that a permission cache is flushed eventually. In this case, temporal logic would be required to express the policy formally. In particular, one would need the "eventually operator" to state the policy correctly. If the policy requirement held that the permission cache would be flushed within ten seconds, then a real-time logic would be required. Using either of these logics increases the level of difficulty of expressing the policy.

Finally, since an adaptive security policy requires greater attention to the timing of actions taken by various system components, especially during transitions, the policy may need to address a finer degree of atomicity. While step-wise refinement might allow one to manage greater complexity in general, by increasing the granularity of detail addressed in the policy, the policy will necessarily become longer and less comprehensible in global terms.

### 2.6.2 Formal Specifications and Proofs

To attain the A1 level of evaluation as specified in the Orange Book [5], it is necessary to formally specify the TCB. Furthermore, the Life-Cycle Assurance for an A1 system requires that evidence is provided showing that this formal top-level specification (FTLS) of the TCB corresponds to the security policy model. While writing the FTLS and proving that the TCB meets the security policy requirements based on the FTLS are two separate tasks, we shall discuss the two together because of their close relation; formal proofs are not required without the FTLS and vice versa.

The FTLS describes the actions of the TCB. Not only do the set of mechanisms which are implemented to enforce a changing policy make the act of specification more difficult, the degree to which one models certain actions on the system as atomic may need to be readdressed. For a system with a static policy it may be acceptable to model an action as atomic when it is in fact not implemented as an atomic action if the difference is not visible to the security policy. However, if the policy can change, then it may no longer be acceptable to model certain actions atomically.

Specifically, there are issues related to cache-flushing as described briefly in Section 2.5.1 and in more detail in [22]. For example, if the policy is changed and cache-flushing is done immediately, then the model must reflect a change in policy as an atomic action. However, if cache flushing is done with some delays, the FTLS must attempt to model the delayed behavior of the system accurately. For some specification languages this may in fact be impossible to model correctly. The temporal coordination of components of the system is clearly more difficult to express than what is required for a system in which one only needs to show that certain rules are satisfied for every system transition. For a policy which can be expressed using real-time logic, the same real-time logic could be used to write formal specifications which can effectively describe the behavior of the system and will support proofs of the requirements, but this still introduces other complexities.

The system as described by the model enforces a set of rules, and the proofs intend to show that those rules match the rules laid down by the policy. Thus in particular when writing proofs and addressing the issue of cache-flushing, assuming that the FTLS is an accurate description of the TCB, one must ask whether the cache represented in the model represents the security policy. If the security policy clearly states how policy transitions ought to be handled, and the model accurately and clearly reflects the TCB and how it handles cached permissions, proof efforts may be straight-forward, but those may be optimistic assumptions.

If the formal policy model requires temporal logic, then so do the proofs. In particular, the notion of eventuality (the permission cache is eventually flushed) poses certain difficulties. First of all, certain "fairness" assumptions would be required to insure that whenever the system has a choice about processing several requests (non- determinism), the system executes each request sometime (i.e., "not never"). Proof rules for reasoning with eventuality can be tedious to apply.

Since an adaptive policy may actually be the implementation of two or more separate policies, then there could be multiple sets of proofs for each separate policy. In addition to the separate sets of rules enforced by each policy, there would also be proofs showing that the policy changes were correct. For a set of $n$ policies, it may be the case that one set of proofs for each policy and one set of correctness proofs for all transitions between them is required. However, if the correctness arguments for transitions depends on the initial and final rule set of the transition, there would be $n(n-1)$ transitions. This represents a substantial increase in the level of effort required for proofs.

### 2.6.3 Spec-to-Code Analysis

When the actions of the TCB are complicated by the policy enforcing mechanisms and the coordination of the components in the system, the task of verifying the correspondence of the specifications to the code will become more complicated just as the task of writing specifications does.

Furthermore, as discussed in the previous section, the complexity of the specifications may depend on the type of logic required to specify the system. Systems which require temporal or real-time logics for specification make performing spec-to-code analysis inherently different from the analysis performed for a system which does not. Again, the issue of eventuality and accompanying fairness assumptions may be difficult to verify.

### 2.6.4 Covert Channel Analysis

There are two approaches for performing covert channel analyses: formal and informal. Formal methods of covert channel analysis are performed when the security policy includes includes statements such as nondeducibility or noninterference security (nondeducibility security was introduced by Sutherland in [26] and non-interference security by Gougen and Meseguer in [11]). Informal methods of Covert channel analysis may employ the Shared Resource Matrix (SRM) Methodology. For neither informal nor formal analyses does the current state of the art encompass adaptive security policies.

The current experience with noninterference analysis has been restricted to deterministic systems with atomic requests. There have been a number of proposals for non-deterministic systems but little practical experience. Experience has been limited to fixed policies. At this time, there is really no theory for how to conduct such an analysis for an adaptive security policy.

As for informal methods, these are essentially manual procedures which are error prone. The additional subtleties of an adaptive policy would increase the likelihood of errors creeping into the analysis. Where there is tool support for conducting informal analyses, the policies have been assumed to be static.

The basic problem with which either method must struggle is the definition of a covert channel for an AIS with an adaptive security policy. Generally covert channel analyses are concerned with MLS policies and the flow of information downward in level. For policies using Type Enforcement, a covert channel is a little different; however, we are simply concerned about client subjects being able to observe information to which they are not authorized through the security policy, whether its rules are determined by MLS or Type Enforcement. Thus, a covert channel in the context of Type Enforcement would be the unintended flow of information across domains by authorized operations.

For example, suppose that an AIS is operating under a set of policy rules $P_1$ under which a subject operating in domain $D_1$ modifies an object of type $T$ using operation $op_1$ and that another subject operating in domain $D_2$ may observe an object of type $T$ using operation $op_2$. Suppose that $op_3$ is the operation that changes the security policy being enforced by the AIS from policy $P_1$ to a new policy $P_2$ where $P_2$ disallows the flow of information from $D_1$ to $D_2$. Then, does the sequence $\langle op_1, op_3, op_2\rangle$ represent a covert channel? Similarly, if $op_4$ changes the policy from $P_2$ to $P_1$, does the sequence $\langle op_1, op_4, op_2\rangle$ represent a covert channel?

Perhaps the point of view of individual sets of policy rules $P_1$ and $P_2$ are simply inadequate to answer the question, and what is required is a super-policy which declares what is allowed or denied at the transitions between them. For example, a relaxed super-policy for $P_1$ and $P_2$

might say that any information flow over a period of transition from the relaxed policy $P_1$ to the stricter policy $P_2$ is allowed. Whereas, a stronger super-policy might be more concerned with contamination (e.g., contamination of objects of type $T$ by subjects in domain $D_1$) and deny certain types of information flow over a period of transition from the relaxed policy $P_1$ to the stricter policy $P_2$. It is not clear what extra level of effort is required for creating a super-policy which would define what covert channels are under an adaptive policy.

### 2.6.5 An Example Application

A stated goal of this report is to analyze and test an application for adherence to security requirements during a policy transition period. The target application is the DTOS medical demonstration which is described in full detail in the DTOS Demonstration Software Design Document [23]. The DTOS prototype is described in greater detail in Appendix A.

The medical demonstration consists of a database server which manages a set of patient records. The database is accessed by various users through client subjects; each client operates in a domain representing the users' functions as physicians, nurses, administrators, etc. These clients do not access the database manager directly but make requests for information through a front end. The front end in turn consults the DTOS Security Server which checks permissions for each client to various aspects of a patient record. The conceptual design of the medical demonstration is illustrated in Figure 2-4.

The point of the original demonstration was to show how applications could be built on top of the DTOS platform, and using the feature for loading (or reloading) the security policy, permissions for the application could be added to the existing platform permissions. Since another set of policy permissions can be loaded at any time without restarting the system, we will define a second set of permissions and discuss the ramifications on assurance tasks from changing from one set of permissions to the other.

The comments made in the preceding subsections were offered as a general discussion of the issues faced when attempting to assure a system with an adaptive security policy. The following subsections provide a concrete system which has an adaptive policy. The following analysis parallels the general discussion, but specializes it to this particular case.

2.6.5.1 The DTOS Medical Demonstration As shown in Figure 2-4 client tasks send requests to manipulate and/or access patient data to the Database Server via the messaging mechanism in Mach. The Demo Exec starts up the Client and Database Server tasks, then controls the operations of the demo based on user input.

Each patient record contains three types of information as illustrated in Figure 2-5: administration, billings and medical. Each kind of information has a set of services associated with them.

- A patient record can be created and deleted
- Admin data can be read and modified
- Billings and medical data (diagnosis and vital signs) can be read, modified and appended.

Access to these services are determined by the relationship between the client's security context[4] and the security context associated with the Database Server's service port.

[4] On DTOS the *security context* is the set of attributes of a subject or object required to make security decisions. In the DTOS medical demonstration the primary attributes are the domain of a subject and the type of an object for a policy employing Type Enforcement.

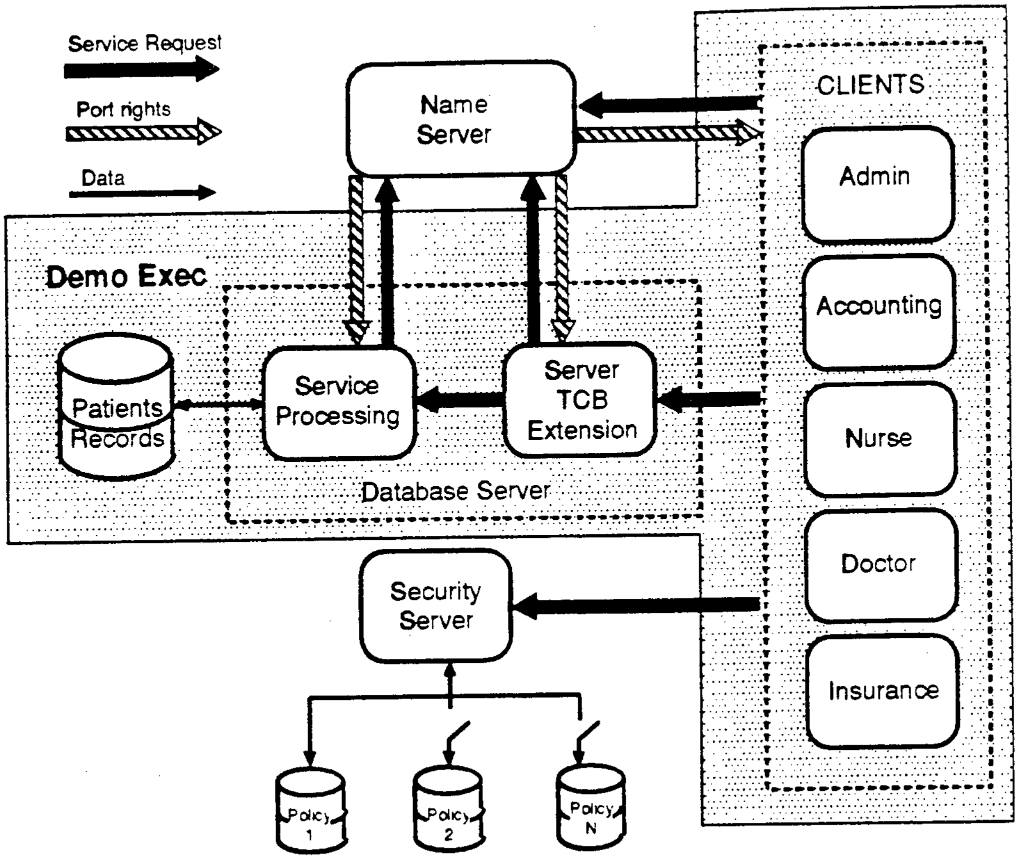


Figure 2-4: Demonstration System

The Database Server interacts with the Name Server to make a send right to its service port accessible to potential clients. The clients obtain a send right to communicate with the Database Server by interacting with the Name Server.

The Database Server TCB Extension (TCB-E) is the part of the Database Server which checks that specific clients have the required permission to the service operations being requested. This part of the Database Server is run in a separate task to demonstrate a stronger separation of the TCB permission checking from the larger more difficult to assure body of software that may make up a real Database Server. The Database Server TCB Extension makes its service port visible to all potential clients via the Name Server.

The Database Server Service Processing is the element of the Database Server which actually carries out the service requests made by clients indirectly through the Database Server TCB Extension. It also makes its service port visible to potential clients through the Name Server.

The DTOS Demonstration client is a generic client executable that is instantiated in multiple tasks in specific domains representing the various types of clients. In the demo, five client domains are implemented: Admin, Accounting, Doctor, Nurse and Insurance. When it is clear, we refer to clients and domains interchangeably. For example, a client task whose domain is Admin, is generally referred as the Admin client. Each client task's domain determines the kind of Database Server service requests it is allowed to send to the Database Server via the TCB-E service port.

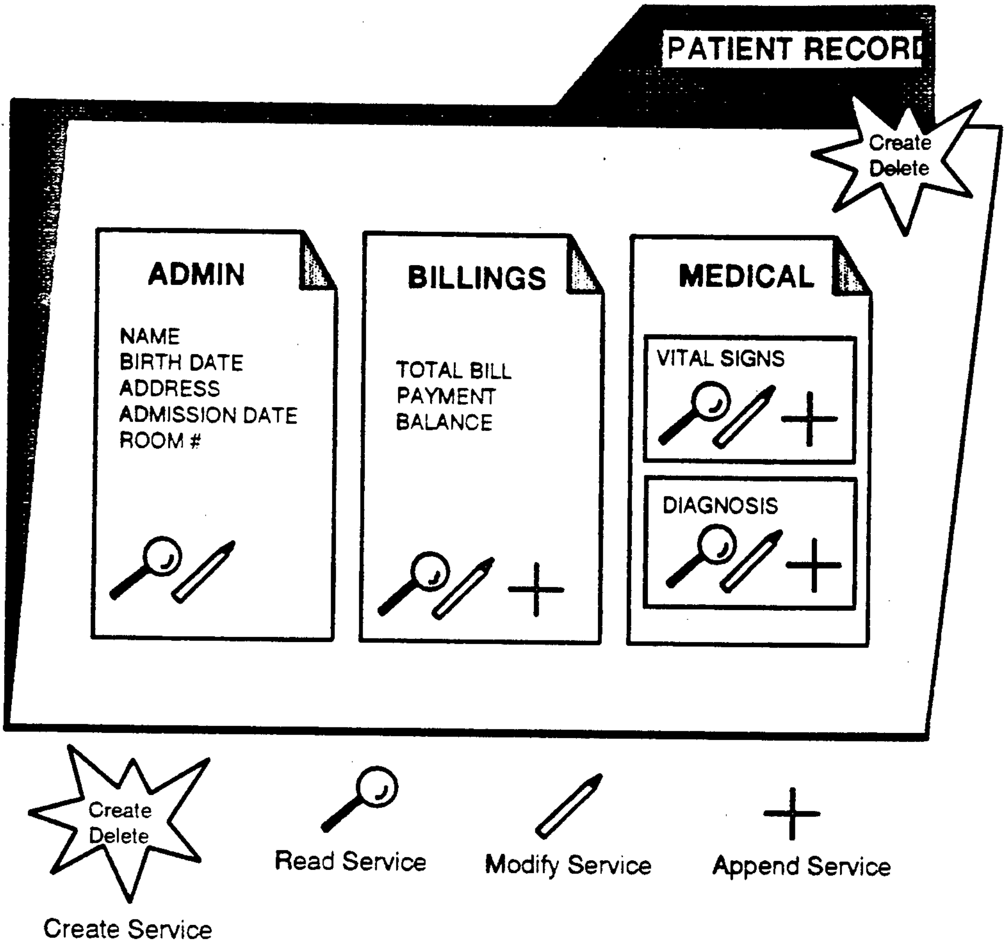


Figure 2-5: A Patient Record

2.6.5.1.1 DTOS Demonstration Security Policy A security policy for the DTOS Demonstration must define the security contexts associated with each subject and object used in the system. The policy is based on SCC's Type Enforcement security policy. The original database security policy for the DTOS Demonstration is summarized in Table 2-5, and a second policy is summarized in Table 2-6. The letters indicate the services that the domains are allowed to request through the database request port. The data which the services process are also shown in the columns. The append permission is a more restrictive form of the modify and some clients may have permission to the former but not the latter.

Under both policies, medical information is restricted to doctors and nurses, billings information to accounting. Under Policy 1, general information is more widely available; while under Policy 2, accounting and insurance clients are no longer able to access billing and administrative data while nurses are given greater permissions to create records, modify administrative data, and append diagnoses.

The rationale for the second set of permissions is that there might be periods, say at night-time or over holidays, when normal administrative tasks are not only not expected to be performed, but to maintain data integrity, no access to these portions of patient records is allowed. Furthermore, since hospital staff may be reduced over these periods, greater accesses

Table 2-5: Accesses to Patient Database Services: Policy 1

| CLIENT DOMAIN | Record | Admin | Billings | Vital | Diagnosis |
|---|---|---|---|---|---|
| Administrator | CD | Ra Ma | Rb | — | — |
| Accounting | — | Ra | Rb Mb Ab | — | — |
| Doctor | — | Ra | — | Rv Mv Av | Rd Md Ad |
| Nurse | — | Ra | — | Rv Mv Av | Rd |
| Insurance | — | Ra | Rb | — | — |

Table 2-6: Accesses to Patient Database Services: Policy 2

| CLIENT DOMAIN | Record | Admin | Billings | Vital | Diagnosis |
|---|---|---|---|---|---|
| Administrator | C | Ra Ma | — | — | — |
| Accounting | — | — | — | — | — |
| Doctor | — | Ra | — | Rv Mv Av | Rd Md Ad |
| Nurse | C | Ra Ma | — | Rv Mv Av | Rd Ad |
| Insurance | — | — | — | — | — |

Patient Database Services

| | | | |
|---|---|---|---|
| CD - | Create/Delete record | Rd - | Read diagnosis |
| Ra - | Read admin | Md - | Modify diagnosis |
| Ma - | Modify admin | Ad - | Append diagnosis |
| Rb - | Read billings | Rv - | Read vital signs |
| Mb - | Modify billings | Mv - | Modify vital signs |
| Ab - | Append billings | Av - | Append vital signs |

must be given to nursing staff who might be in relatively greater supply.

Figure 2-6 shows the Admin client making successful requests to create a record, and to modify admin data while the Doctor and Insurance clients requests for the same services fail due to lack of permissions. The shaded areas in the access vectors indicate permissions which the clients do not have, but are shown for completeness.

Figure 2-7 shows another example of service controls in which the Nurse client in now allowed to append diagnosis data which could only be done by the Doctor client under Policy 1. In addition, the Accounting client fails on all requests, even on the request to modify billings to which it has access under Policy 1.

2.6.5.2 Ramifications for Assurance Tasks In the following subsections, we will echo the general comments made in Section 2.6 regarding the impact of non-tranquility on specific assurance tasks providing specific comments as allowed by the example application.

The security policy for this specific application is relatively simple. The objectives of the application are to release portions of patient records only to those people who have a need

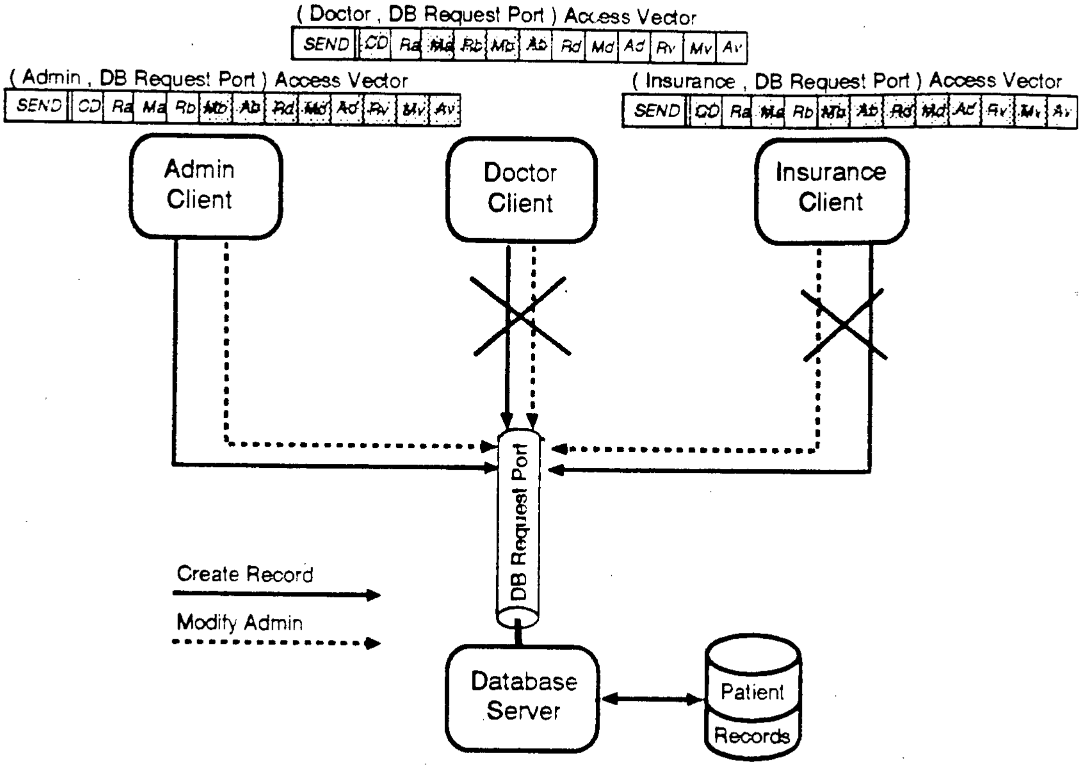

( Doctor , DB Request Port ) Access Vector
SEND CD Ra Ma Rb Mb Ab Rd Md Ad Rv Mv Av
( Admin , DB Request Port ) Access Vector
SEND CD Ra Ma Rb Mb Ab Rd Md Ad Rv Mv Av
( Insurance , DB Request Port ) Access Vector
SEND CD Ra Ma Rb Mb Ab Rd Md Ad Rv Mv Av
Admin Client
Doctor Client
Insurance Client
DB Request Port
Create Record
Modify Admin
Database Server
Patient
Records

Figure 2-6: Service Checks for Admin, Doctor and Insurance Clients Under Policy 1

to read them and to allow a controlled group of people access to create, modify, append, or destroy records. It must be assumed that users and client subjects which have access to modify records will act appropriately. Thus, by controlling the privilege to modify, we may argue that the integrity of the database can be maintained. Audit records can be used to provide accountability for inappropriate modifications of data records.

As discussed in Section 2.6, the effect of introducing the second policy for the application depends on how the transition between the two policies is implemented. For this application it is probably acceptable to allow for some delays in the transition between Policy 1 and Policy 2; however, this probably has the greatest effect on the assurance tasks. Flushing cached permissions immediately upon the change of policy reduces the impact.

2.6.5.2.1 Policy Modeling The security is more than just the collection of tables represented by Figures 2-5 and 2-6. Immediate transitions from one policy to the other, makes the policy for the transition easier to model. This is the current implementation. As previously noted, DTOS supports a mechanism for reloading the security policy, and this automatically flushes all cached permissions.

As discussed in Section 2.6, if the implementation and the security policy allow for the transition to occur at any future time rather than immediately, then the policy would have to modeled using temporal logic.

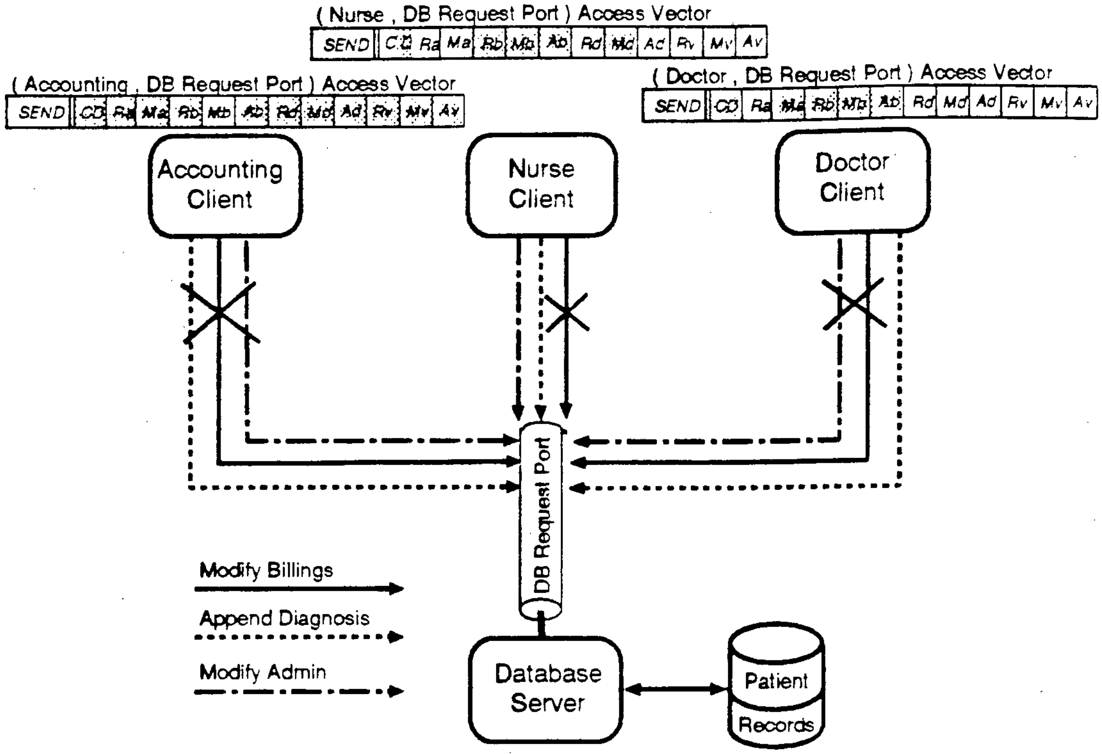

( Nurse , DB Request Port ) Access Vector
SEND CD Ra Ma Rb Mb Ab Rd Md Ad Rv Mv Av
( Accounting , DB Request Port ) Access Vector
SEND CD Ra Ma Rb Mb Ab Rd Md Ad Rv Mv Av
( Doctor , DB Request Port ) Access Vector
SEND CD Ra Ma Rb Mb Ab Rd Md Ad Rv Mv Av
Accounting Client
Nurse Client
Doctor Client
DB Request Port
Modify Billings
Append Diagnosis
Modify Admin
Database Server
Patient
Records

Figure 2-7: Service Checks for Accounting, Nurse and Doctor Clients Under Policy 2

2.6.5.2.2 Specification and Proofs There are a small number of components to this application, in fact the specifications for this example could be limited to the Database Server TCB Extension and a generic client subject. The actions of the Name Server and to some extent the Security Server could be abstracted away.

The main issue is whether permission caching would need to be modeled. Since permissions are flushed immediately, one might not even model the caching of permissions and model the system as if the Database Server always queried the Security Server for access permissions. A number of the details can be abstracted away. If this is the case, then the specifications are quite simple, perhaps no more complicated than for a single policy.

However, if an alternative implementation allowed delays in cache flushing, then not only would the permission cache need to be specified, it would have to be modeled accurately using a temporal or real-time logic.

2.6.5.2.3 Spec-to-Code Analysis Again, the main issue for spec-to-code analysis is whether permission caching is modeled. If the specifications are simple and many of the details are abstracted away than the spec-to-code analysis would also be relatively simple. If specifications are written using a temporal or real-time logic, then the spec-to-code analysis becomes more complicated commensurate with the added complexity of the specifications.

2.6.5.2.4 Covert Channel Analysis The policy for this application does not support MLS access controls; so as discussed in Section 2.6.4, we must be careful to state what a covert channel is in this context. We are concerned about client subjects being able to observe information to which they are not authorized through the Type Enforcement policy, and we must define what flow of information across domains by authorized operations ought to be allowed over periods of transition.

In the case of the policies defined for the medical example, Policy 1 allows the Insurance client to read administrative records, and no one who may modify the administrative records (Administrator clients) can read either the diagnosis or vital signs. However, under Policy 2, the Nurse client may read the diagnosis and vital signs and may also modify the administrative records. The potential for information to flow from the diagnostic information to the domain for the Insurance client exists when the policy changes from Policy 2 to Policy 1.

If the super-policy holds that the Insurance client shall have no access to diagnostic information, then a covert channel may exist. Thus, the covert channel analysis would have to examine how the Nurse client may modify the administrative records and the rate at which information may flow from diagnostic records to administrative records. Since the policy transitions would occur at long intervals, at most two per day, the capacity for this channel is probably very low, especially if the Nurse client is highly constrained in the ways that it may modify the administrative records, minimizing the number of characters that can be transmitted. However, even low bandwidth could have potentially devastating results. The message "John Doe has cancer" only needs to be sent once to cause great harm. As with many other security and safety applications, some events are unacceptable even if their likelihood is so small that the expected loss, as calculated by statisticians and actuaries, seems acceptable.

## 2.7 Summary

Section 2.4 surveyed the range of tranquility assumptions that one can make in formulating a security policy and their utility. However, in order to provide the adaptive security policies required by the scenarios described in Section 2.3.4, a number of these tranquility assumptions must be discarded. Section 2.4.3 discusses these specific assumptions and for each set of tranquility assumptions determines which scenarios would need to discard that set of assumptions.

While other work on dynamic lattices (see [13]) has laid down the formal properties that dynamic lattices must satisfy, a system implementing dynamic lattice must be able to represent them. Section 2.5.2 makes two specific proposals for representing and managing dynamic lattices. The first solution, which employs the use of additional, artificial categories, could be useful for small lattices in which the location of new lattice points can be anticipated. The second solution, which uses the local structure of the security lattice, is more practical for larger lattices and for situations in which the inclusion of lattices is unanticipated.

Section 2.5.3 proposes a High-Water Mark Confidentiality Audit Policy. When attempting to recover from the relaxed policy, the mechanisms described could be used either as an auditing tool to determine if subjects and objects had really be contaminated with high level information, or it could be used for the sake of mandatory access controls preventing the possibility of further contamination from occurring.

Section 2.6 discusses the ramification of the loss of tranquility on specific tasks associated with formal assurance. Although, there is some theoretical work on the subject ([10] and [9]), there are still some large steps to take to fully comprehend the nature of the impact that an adaptive security policy has on the assurance evidence for an application or system.

*Section* 3

# Audit

## 3.1 Introduction

Prior work on adaptive security (see [22]) discussed the use of auditing to assist in recovery from a period of relaxed security. To facilitate this recovery, tracing of information flows must be performed. Once this goal is accomplished it is possible to know what objects in the system have been contaminated with higher security level data.

Investigation of audit logs revealed several deficiencies in the DTOS auditing information that prevented us from using audit logs for tracing information flow. The major problem was the inability to relate the audited fine grained permission check of the microkernel, to high level activities of the system. For example, it is impossible to discern that a file was being opened via audited microkernel permission checks. The other problem was that not enough information was presented in the audit data to be able to relate a set of audited permission checks to a particular chain of execution. [5] We addressed these deficiencies by modifying the DTOS prototype to provide additional data on each audit event, and by auditing service request messages and their contents.

As stated, DTOS audit logs do not provide the information needed to be able to relate one audited permission check to another audited permission check in a single chain of execution. The approach taken to rectify this situation under the ASP2 program is to maintain a tracing identifier that is created when *user* domains[6] send a message. This tracing identifier (referred to as TID, hereafter) is included in the audited information for each audited permission check. If the receiver of the message is not another *user* domain, the TID for any messages the receiving thread sends, will be the TID from the last message received by that thread. The TID is used to group a set of audited permission checks to one chain of execution.

The other shortcoming is that the permission checks are too low level to be able to relate to functional operations. The majority of the interesting events take place in system servers running in user space. To address this shortcoming under ASP2, the kernel was modified to snoop all messages for a configurable set of service requests. When such a service request is found, an audit event is generated with a user defined set of the service request parameters included in the audit data.

## 3.2 Logical Groupings of audited permission checks

To facilitate grouping of microkernel audit data related to a particular system activity, a tracing identifier (TID) was added to the microkernel thread structure. A TID value is set in one of two ways depending on the domain of the task. If the domain is considered a *user* domain, the TID is set to a unique value when the thread sends a message. Otherwise, the TID is set upon receipt of a message. In this latter case, the receiver's TID is set to the TID of the sender's thread.

[5] A chain of execution is all the processing that takes place in order to satisfy a single system service request, including processing that takes place in several different servers.

[6] A *user* domains are application layer domains. *System* domains are for servers and the microkernel.

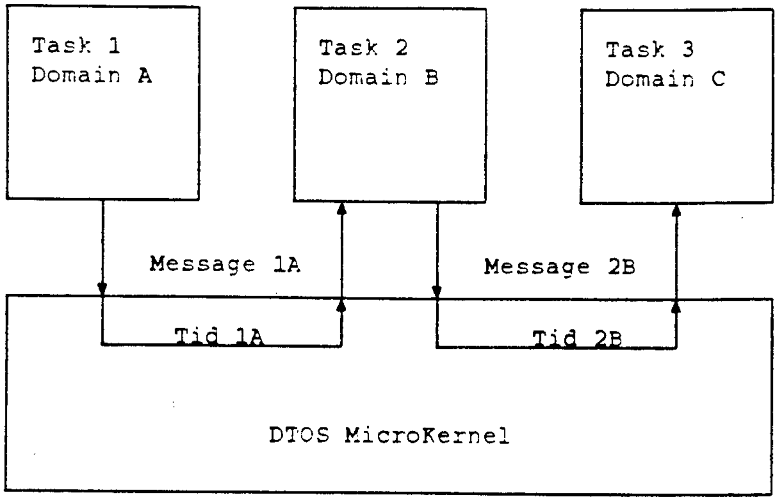


Figure 3-8: Tracing Identifier Flow

In Figure 3-8, Thread 1 sends a message (Message 1A) to Thread 2. Assuming Domain A is defined as *user* domain, a new TID (TID 1A) is created and assigned to the thread structure responsible for sending the message. TID 1A is also carried along in the message to Thread 2. The thread in Thread 2 receiving Message 1A has its TID value set to TID 1A. Next Thread 2 sends a message (Message 2B) to Thread 3. If Domain B is not defined to be *user* domain, then the TID assigned to the message will be the TID of the thread sending the message. In this case that TID would be TID 1A ie, TID 2B would be the same as TID 1A. The TID value for the currently executing thread is then appended to the audit data sent to the Audit Daemon. Thus the audited permission checks relating to a specific *user* domain service request will all bear the same TID even as the resolution of that service request migrates through several different tasks/servers.

The specification of which domains are *user* domains is performed via a new kernel interface on the host port. This interface, *host_audit_control*, allows applications to set and reset a domain's *user* status. By default, domains are considered not to be *user* domains. The security of this interface is controlled via the usual DTOS means. Thus only specific domains (auditing domain for example) should be permitted access to this interface.

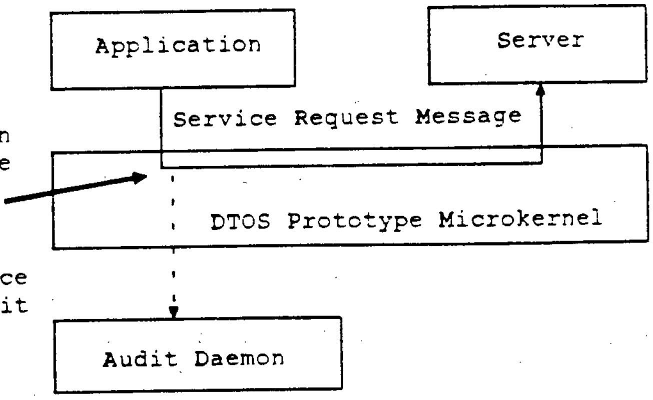
While this method for grouping permissions is sufficient for the majority of cases, servers that circumvent the MACH IPC system for passing processing control to another thread, must use a manual means for managing TID values. In order to facilitate migration of TID values for a chain of execution that include non IPC messaging (ie, via shared memory) a new service was added to the thread port. The service *audit_thread_tag* is used by the server to obtain the TID value from thread structure for the current chain of execution. The server must then inform the new thread in the chain, such that the new chain may also invoke the *audit_thread_tag* request to set its TID value. The *audit_thread_tag* request should only be permitted for the domains in which the multithreaded servers reside.

No attempt was made to modify the existing multithreaded DTOS servers (Lites for example) to follow the rules listed above. Such an effort was beyond the scope of this research effort. Attempts to modify multithreaded servers might prove to be difficult depending on how clearly thread handoffs are marked in the source code, and how centralized are the implementations of the thread handoff mechanism.

## 3.3 Auditing of system servers via microkernel snooping

In a server/microkernel architecture such as DTOS, the majority of the security critical events are performed by system servers rather than the microkernel. An example of this is the Unix operation of opening a file. In the DTOS prototype, this operation is handled by sending a file open service request to the Lites Server. One approach for providing audit information on similar higher level activities, is to modify each server to interact with the Audit Daemon for each service provided. Another approach is to centralize the auditing of all service requests in the microkernel. The former approach is the ideal auditing mechanism in that the server can supply the audit daemon with whatever information the server deems appropriate. The drawback of the former approach is that all service request routines in every server need to have auditing code inserted. The latter approach provides a flexible system that lends itself to easy integration of new servers. The drawback of the latter approach is the system processing overhead of snooping each message.

In order to facilitate auditing of services provided by system servers without modifying the servers, the DTOS microkernel was modified to snoop messages sent to servers for service requests. The implementation allows an external agent to inform the microkernel that messages sent from a task with a specified subject SID with a specific message identifier[7] , sent to a port with a specified SID , will trigger an audit event. This audit event will have a specified subset of the parameters decoded and sent as part of the text of the audit message. The subject SID, message identifier, and port label may all be wildcarded.



When the DTOS prototype receives a message from the application, it compares the Application SID, Server SID, and the service request number (message ID) against a list of auditable services. If the service requests matches an audit event is generated.

Figure 3-9: Snooping of Service Requests

The interface *host_audit_control* is used to add and remove entries from the list of messages that are snooped. The performance impact of the message snooping increases with each entry in the list. Thus if the auditing policy requires a large list of services to be audited, then either a more efficient algorithm for pattern matching of the message would need to be developed, or the auditing should be moved into the service routines within each server.

## 3.4 Implementing adaptive security with the assistance of auditing

By itself, the DTOS Security Server is limited to permission computations, accessible system state, and human interaction in order to trigger policy adaptations. However, with some minor

[7] The Mach IPC mechanism includes an integer value called the message ID. This value is used by the Mach Interface Generator (MIG) to differentiate service routines offered on a single port

modifications to the Security Server and the Audit Daemon, the range of triggers for policy adaptations has been considerably expanded. The modifications allow the Security Server to configure the Audit Daemon via a port advertised on the name server. The Security Server can also supply a send right to a port on which it will be notified when the configured audited event takes place. For example, given a Chinese Wall (see [20]) type of security policy, where a user is permitted access to either of two files, but once access to one of the files is performed, access to the other of the two is restricted. The Security Server could configure the audit daemon to audit accesses to both files and to notify the Security Server when such an access takes place. The Security Server, upon receiving such a notification (or trigger), would adapt the security policy such that access to the other file is restricted.

At this time, limited controls on the use of this interface have been implemented, but there is some security risk that the audit configuration port could be used to create an audit configuration that cannot be physically supported (for example, auditing all permission checks generates enough audit messages that it overflows the audit port). Or the interface could be used to create an audit configuration that supersedes the intended audit functionality. The implemented controls involve the Audit Daemon returning a key back from every audit configuration change. This key must be supplied in order to undo the configuration change.

The implementation of the audit configuration interface is limited to changing the auditing of service request messages. There is no reason for the Security Server to have the Audit Daemon monitor permission checks on behalf of the Security Server when the Security Server can set the cacheable flag in such a way as to monitor permission checks itself.

## 3.5 Using Auditing to recover from periods of relaxed policy

With the enhancements to the DTOS auditing mechanism described above, the capability to generate an audit trail with enough contextual information to discern information flows has been added to the DTOS prototype. However, in order to make use of this information the auditing mechanisms must be configured to look for the significant information flows. To address this requirement, a rudimentary scheme to automating the auditing configuration was implemented. This scheme involved comparing two security policy files. The automation tool generated an audit configuration for the difference between the two policies. A more involved scheme would involve an automation tool that is aware of data flow significance of each permission pair and would generate an audit configuration that only audits significant differences.

Another means of monitoring leakage of information from a high security level to a low security container would be to modify the Audit Daemon to adapt its audit policy on the fly, based on prior audited events. While this modification was not performed as part of the ASP 2 program, it is easy to see how it could be utilized. With an Audit Daemon modified in such a manner, the Audit Daemon would recognize when high security data was placed in a low security container. At this point the Audit Daemon would reconfigure the audit policy to also monitor information flows from this low security container, as it is now contaminated with high security data.

## 3.6 Conclusions

It is obvious that in order to evaluate and recover from a period of relaxed security policy, there is a need to track information flows that take place during this period. Auditing can be a useful tool for use by the Security Server and by security administrators to monitor system activity and information flows. On the DTOS prototype the Security Server implements the policy for the DTOS microkernel and to some extent the Lites Unix Server. Any service provided by a system server that is not controlled by the Security Server cannot be monitored for information flows by the Security Server. In this case, if the service can be monitored via auditing, the Security Server could use the Audit Daemon to monitor the information flows even though the Security Server cannot directly control the interface. In addition, tracing identifiers permit the differentiation of information flows between applications and system servers. With the modifications to the DTOS prototype to permit microkernel snooping of service requests and the addition of tracing identifiers, it is feasible to utilize audit data to assist in recovery from a period of relaxed security policy. The tools and modifications to the DTOS prototype will be made available in the next release of the DTOS prototype. Future work in this area could be oriented toward automating recovery from the period of relaxed policy, given the information flows presented from the audit log. Other avenues of investigation could include dynamic audit policies and further automation of audit policy generation.

*Section 4*

# Security Database Tools

## 4.1 Introduction

This section describes the design and implementation of tools for constructing security databases.

For any automated information system (AIS) which is expected to enforce a security policy, the security policy must be encoded in a database that the AIS can read and interpret. For the DTOS prototype it is the Security Server that defines the policy, and makes security computations on behalf of the microkernel and other servers. The Security Server defines the security policy from the time that it is initialized by reading its security database. With an adaptive security policy, there are two or more policies under which the AIS may run, and therefore it is necessary to construct two or more databases to define the security policy after each transition in addition to the initial definition. Although the similarities between the two policies may be greater than the differences, it can be a difficult task to manage the information that must change from one policy to the next. Maintaining a large database by hand, using only a text editor for example, is prone to error. The alternatives to maintaining a database by hand include generating the security databases by compiling a text based specification language or by encapsulating the specification in a tool. The approach taken on this program is to encapsulate the specifications in a security database tool with a graphical user interface. This approach was chosen over creating a formalism for high level security policies as a result of usability and implementation concerns.

The design of a database tool must meet the following specifications:

- It must allow the user to specify the policy in real-world terms; i.e. it must help the user map her organization's security policy to the security policy that will be enforced by the AIS.
- It must not prevent the user from effectively controlling the permission set at the lowest levels.
- It must provide access to the Adaptive Policy Mechanisms supported by the Security Server.

The following subsections of this report will discuss the set of existing database tools for the DTOS prototype and the design and implementation of the GUI tools for specifying the database.

Before continuing with the following descriptions, it will be helpful to define clearly two terms that shall be used throughout the remainder of this section: policy files and database files. The first term refers to files created by the security administrator (tool user) to define the security policy. The latter refers to the processed[5] policy files that are the actual input files for the AIS (in the case of the DTOS prototype, the Security Server's security database files).

[5] processed - in that they are automatically constructed from the policy files by the database tools.

## 4.2 Existing database tools

The DTOS prototype is supplied with rudimentary database tools. These tools consist of a Makefile that processes the policy files through the M4 macro processor and some perl scripts. The M4 macro processing step provides a limited capability to logically group permission pairs into functional sets. The Perl scripts perform the task of converting symbolic permission names into bit positions in an access vector.

The are two problems with the existing database tools. First, the macros used to logically group the pairs represent a very high level of abstraction that prevents simple modifications to specific domains. Typically, one must duplicate the entire macro for the specific domains to be modified and change one of the macros to contain the specific modification, while leaving the unmodified macro for use by all other domains. The second problem is the non-obvious syntax of both sets of text files which makes human interpretation of the database files difficult. To address the problems presented above, a GUI database tool was created.

## 4.3 Design and Implementation of database tools

Two approaches were considered for developing database tools to support the Adaptive Security Policies Experience program. The first approach was to develop a formalism for specifying higher level security policies, then create database tools to take the formal policy specification, and convert it into database files for the Security Server. The work in progress at ORA and Secure Computing's TESLA policy specification language are examples of this approach that were investigated for applicability to this program. The work at ORA had not progressed far enough to be available for use, and the TESLA policy specification language was not deemed to be appropriate for adaptive policies.

The second approach was to utilize a GUI database tool that generated security database files. Such a tool obviates the need for the policy writer to learn a specification language. This approach is characterized by the Adage tool set from the Open Group Research Institute. This latter approach utilizing the Adage toolset was the preferred approach to the creation of database tools due to preexisting technology and the ease of creating and editing security policies with a GUI based tool. However, the Adage tool set was in the process of being rewritten during the timeframe of this program. This resulted in the decision to build a GUI database tool set locally. In order to provide a visual interface, a database tool was created to run on a Windows 95 / Windows NT system. This choice was made due to the wealth of development tools available for the Windows platform.

The database tools developed for the DTOS prototype under this program have been designed to meet the following requirements:

1. Permit grouping of permissions into hierarchical sets.
2. Provide an easy upgrade path from the existing DTOS policy files by supporting the existing file format (read only).
3. Support a specific adaptation mechanism.
4. To minimize modifications to the Security Server, the database tools must use a format for the database files compatible with the existing DTOS Security Server. (With the exception of adaptation information)

To meet the first requirement, and to ease the implementation of the second, the GUI tool continues to support the concept of macros, which are the basic building blocks of the security policy. Each modular entity in the security policy should be relegated to its own macro, thus providing the capability to build up a security policy in a modular fashion. The shortcoming of modular construction of a security policy is the need to provide for exceptions to the default behavior of a module. The GUI tool was designed to handle these exemptions by providing the capability to edit a macro and to apply the edited macro either to all invocations of the macro, or to just the current invocation of the macro. The implementation of this feature in the GUI tool was not completed due to time constraints.

The implementation of the GUI tool supports reading the old format of policy files but has its own native format for saving and restoring policy files. The decision to use a native file format for saving policy files was based on the number of unique files and formats that were used to define the DTOS format of policy files. The DTOS format of policy files used eleven files with six different formats. This was consolidated into one file in the native format file.

Two types of policy adaptation mechanisms where considered in developing the database tools, time-of-day-based adaptations and event-based adaptations. A time-of-day based adaptation is an adaptation that takes place at a specific time of day, every day. For example, a bank's AIS may start to restrict account transactions at 5pm. Those restrictions may be removed at 8am the following day. The second type of policy adaptation, event based adaptations, provides for most other types of policy adaptations. An event can be considered to be any form of automated trigger to a policy change. The trigger could be an audit event, a particular permission check, a signal from an intrusion detection daemon, or any other system or security events.

To support these two types of adaptation means, two subframes (or windows) of the GUI interface have been defined. The first subframe is used to generate time-of-day based adaptations. The second is used to define event based adaptations. These subframes require the policy writer to create a macro that defines the policy action which will take place when adaptation criteria is met. When the GUI tool generates the database files, the adaptation information will be included in a new section of the files. The DTOS Security Server was modified to use the adaptation information to generate time of day based adaptations. However, due to time constraints, event-based adaptations were not implemented as part of this program.

The screenshots in Figure 4-10 through Figure 4-15 show various frames of the security database tools.

- Figure 4-10 shows the starting display of the database tool after the database files have been loaded. Note the list of macro invocations in the upper right half of the screen and the base policy SID pairs in the lower half of the screen. The controls on the upper left of the frame control the Type Enforcement primitives and the MLS relations. The controls in the middle of the left side are for controlling policy adaptations.

- The interface used to modify the permissions associated with a given database pair is illustrated in Figure 4-11. The permission names are parsed directly from the *.h files that define the access vectors.

- In Figure 4-12 the interface used to create and edit macros is shown. The parameters to the macro can be referenced by using $1, $2, $3, and $4 for parameters one through four respectively.

- Figure 4-13 shows the specialized form used to create time-of-day based policy adaptations.

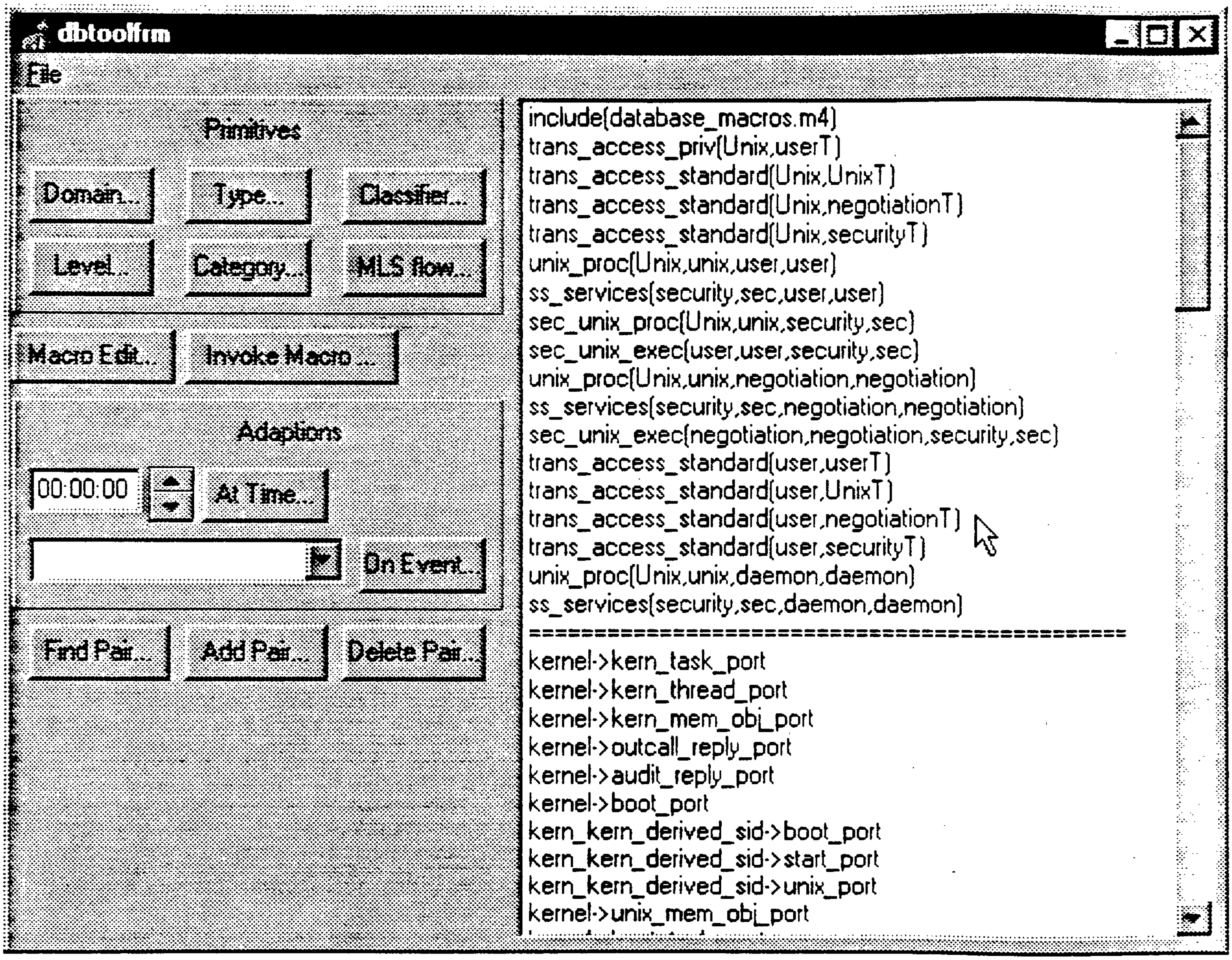
- Figure 4-14 demonstrates the interface used to edit MLS flows associated with specific permission. In the DTOS prototype, each permission may be associated with an MLS information flow. This interface allows the database developer to redefine these information flows, or to specify the information flow for a new permission.

- Figure 4-15 illustrates the interface used to invoke macros.

## 4.4 Conclusions

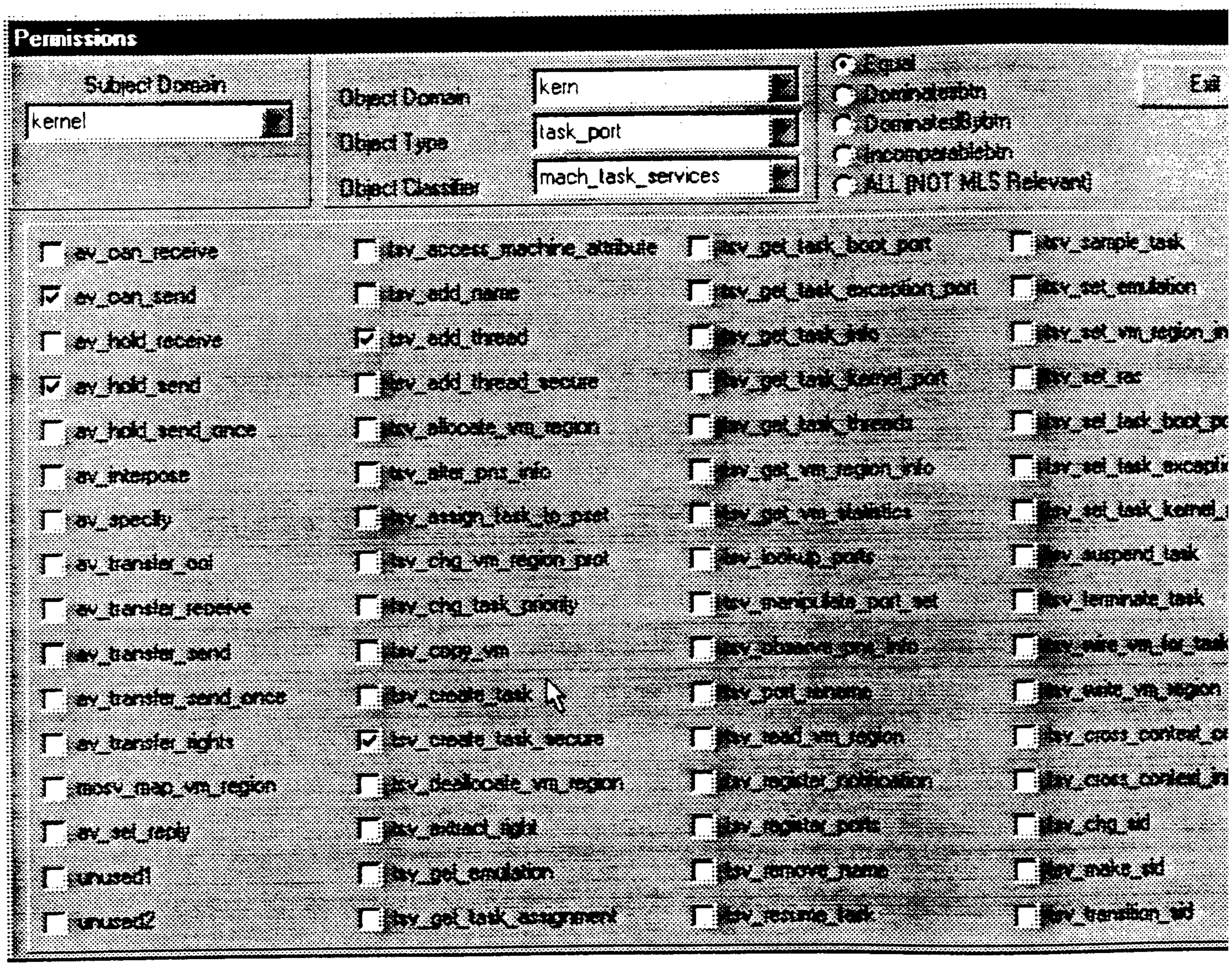
Security policy database files for any moderately complex AIS will require some form of management tools as the complexity of the database files precludes direct human management. The policy management tools designed under this program support a hierarchical system of building up a security policy without preventing variations from the default hierarchy by specific domains. And while the implementation of these tools is tailored for the DTOS prototype, the design is applicable to any Type Enforced[1] security system. These database tools will be made available with the next release of the DTOS prototype.

The next step in the utilization of the GUI database tools would be to regenerate the DTOS policy files from scratch. In the process of regenerating the policy files, special attention would be given to modularizing the policy into a hierarchical set of macros. This new hierarchical set of macros would permit the GUI tool to be more effective in creating policies that incorporate least privilege. Other avenues of investigation could involve unifying the audit policy and the security policy into one policy tool and increasing the range of adaptations supported by the policy tool and Security Server to include event-based adaptations.



```
include(database_macros.m4)
trans_access_priv(Unix,userT)
trans_access_standard(Unix,UnixT)
trans_access_standard(Unix,negotiationT)
trans_access_standard(Unix,securityT)
unix_proc(Unix,unix,user,user)
ss_services(security,sec,user,user)
sec_unix_proc(Unix,unix,security,sec)
sec_unix_exec(user,user,security,sec)
unix_proc(Unix,unix,negotiation,negotiation)
ss_services(security,sec,negotiation,negotiation)
sec_unix_exec(negotiation,negotiation,security,sec)
trans_access_standard(user,userT)
trans_access_standard(user,UnixT)
trans_access_standard(user,negotiationT)
trans_access_standard(user,securityT)
unix_proc(Unix,unix,daemon,daemon)
ss_services(security,sec,daemon,daemon)
================================================
kernel->kern_task_port
kernel->kern_thread_port
kernel->kern_mem_obj_port
kernel->outcall_reply_port
kernel->audit_reply_port
kernel->boot_port
kern_kern_derived_sid->boot_port
kern_kern_derived_sid->start_port
kern_kern_derived_sid->unix_port
kernel->unix_mem_obj_port
```

Figure 4-10: The GUI Database Tool



Subject Domain: kernel

Object Domain: kern

Object Type: task_port

Object Classifier: mach_task_services

● Equal
○ Dominatesbtn
○ DominatedBybtn
○ Incomparablebtn
○ ALL (NOT MLS Relevant)

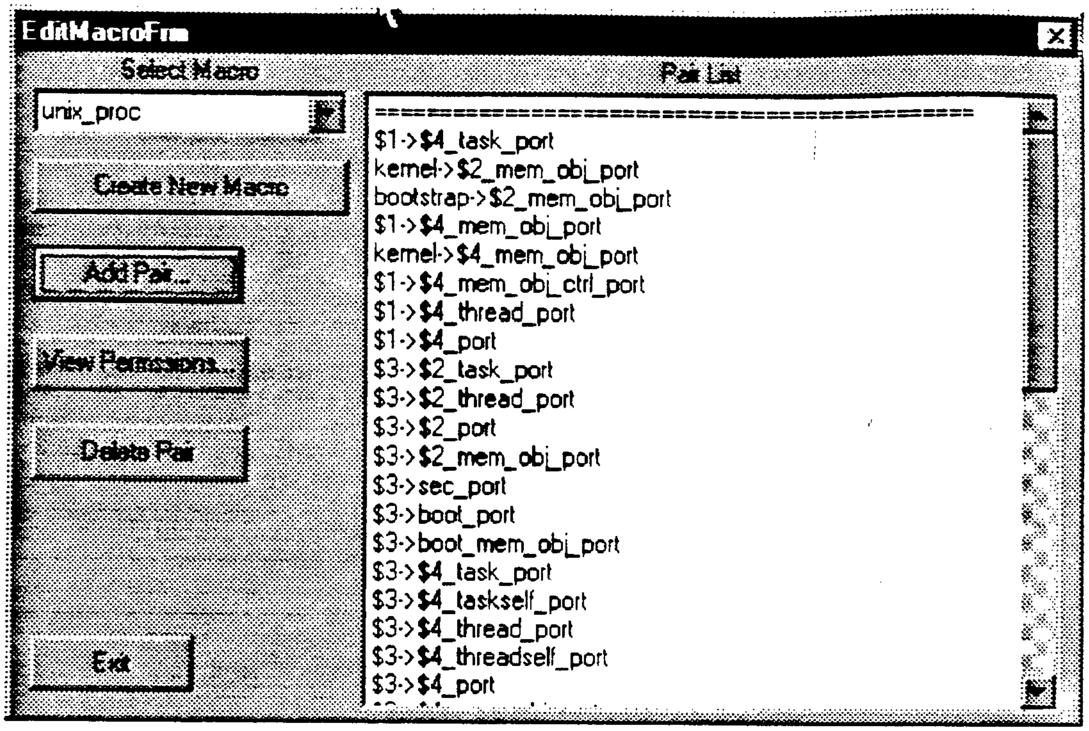
☐ av_can_receive
☑ av_can_send
☐ av_hold_receive
☑ av_hold_send
☐ av_hold_send_once
☐ av_interpose
☐ av_specify
☐ av_transfer_ool
☐ av_transfer_receive
☐ av_transfer_send
☐ av_transfer_send_once
☐ av_transfer_rights
☐ mosv_map_vm_region
☐ av_set_reply
☐ unused1
☐ unused2

☐ tsv_access_machine_attribute
☐ tsv_add_name
☑ tsv_add_thread
☐ tsv_add_thread_secure
☐ tsv_allocate_vm_region
☐ tsv_alter_pns_info
☐ tsv_assign_task_to_pset
☐ tsv_chg_vm_region_prot
☐ tsv_chg_task_priority
☐ tsv_copy_vm
☐ tsv_create_task
☑ tsv_create_task_secure
☐ tsv_deallocate_vm_region
☐ tsv_extract_right
☐ tsv_get_emulation
☐ tsv_get_task_assignment

☐ tsv_get_task_boot_port
☐ tsv_get_task_exception_port
☐ tsv_get_task_info
☐ tsv_get_task_kernel_port
☐ tsv_get_task_threads
☐ tsv_get_vm_region_info
☐ tsv_get_vm_statistics
☐ tsv_lookup_ports
☐ tsv_manipulate_port_set
☐ tsv_observe_pns_info
☐ tsv_port_rename
☐ tsv_read_vm_region
☐ tsv_register_notification
☐ tsv_register_ports
☐ tsv_remove_name
☐ tsv_resume_task

☐ tsv_sample_task
☐ tsv_set_emulation
☐ tsv_set_vm_region_in
☐ tsv_set_ras
☐ tsv_set_task_boot_p
☐ tsv_set_task_exceptio
☐ tsv_set_task_kernel_
☐ tsv_suspend_task
☐ tsv_terminate_task
☐ tsv_wire_vm_for_task
☐ tsv_write_vm_region
☐ tsv_cross_context_c
☐ tsv_cross_context_in
☐ tsv_chg_sid
☐ tsv_make_sid
☐ tsv_transition_sid

Figure 4-11: Permission Modification Frame

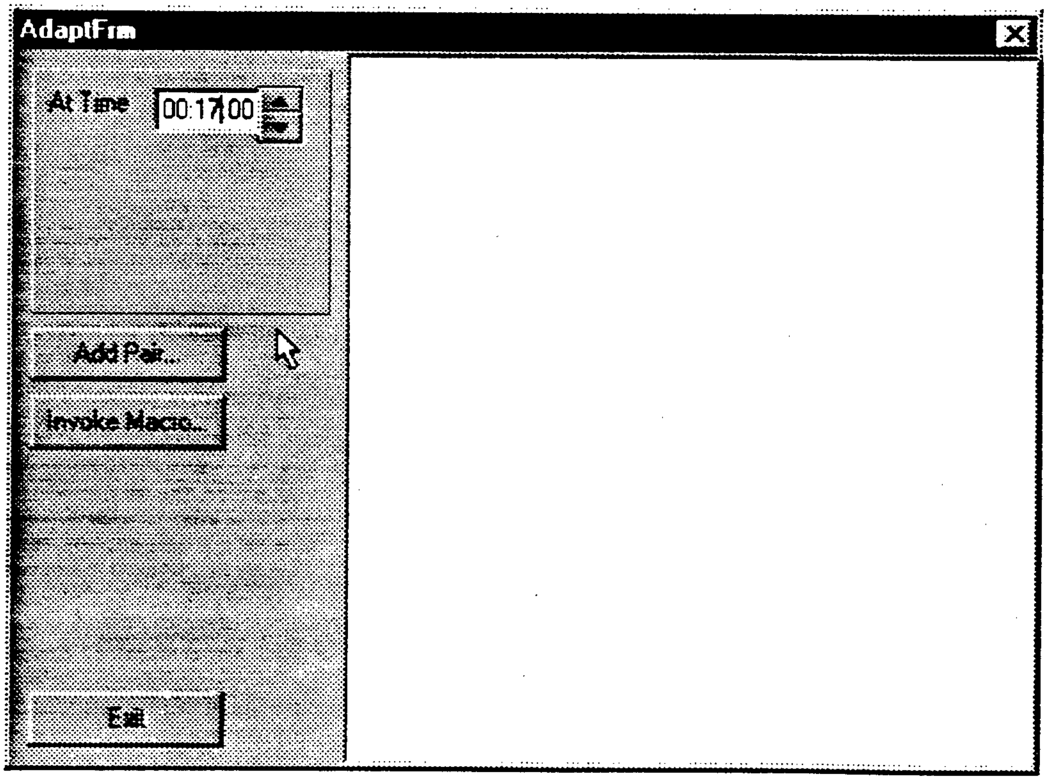


Select Macro: unix_proc

Pair List

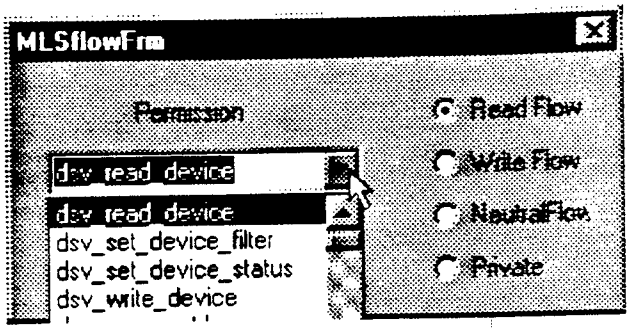
```
========================================
$1->$4_task_port
kernel->$2_mem_obj_port
bootstrap->$2_mem_obj_port
$1->$4_mem_obj_port
kernel->$4_mem_obj_port
$1->$4_mem_obj_ctrl_port
$1->$4_thread_port
$1->$4_port
$3->$2_task_port
$3->$2_thread_port
$3->$2_port
$3->$2_mem_obj_port
$3->sec_port
$3->boot_port
$3->boot_mem_obj_port
$3->$4_task_port
$3->$4_taskself_port
$3->$4_thread_port
$3->$4_threadself_port
$3->$4_port
```

Figure 4-12: Macro Editing Frame



At Time 00:17:00

Figure 4-13: Adaptation Generating Frame



dsv_read_device

dsv_read_device
dsv_set_device_filter
dsv_set_device_status
dsv_write_device

● Read Flow
○ Write Flow
○ NeutralFlow
○ Private

Figure 4-14: Controls for Specifying MLS flows

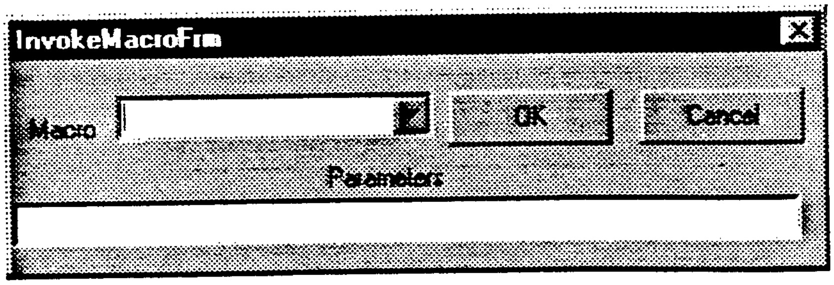


Figure 4-15: Macro Invocation Frame

# *Section* 5 Trade-Off Study

## 5.1 Introduction to the Trade-Off Study

The Trade-Off Study[9] compares four methods for implementing adaptive security policies. Two of these methods were identified in [22] but two of these have been conceived and partially implemented for the sake of the current study.

Several criteria have been identified for the sake of comparing these implementations. These criteria are defined and explained in the following section; however, in order to adequately compare implementations of adaptive security policies, it is important to keep in mind the security and functional needs of organizations that would deploy systems with adaptive security policies. The following several paragraphs outline several examples and scenarios for which adaptive security policies would be implemented and deployed. Some of the following examples are also covered in Section 2.3.4.

The first example of adaptive security consists of organizations that need to change their policies at regular intervals. For example, a bank may have one security policy enforced during business hours and another policy enforced after hours. The business-hours policy would grant broad sets of permissions to various sets of employees in order complete normal banking transactions; however, a more restrictive policy would be in effect after hours to prevent system users from altering banking data in unintended ways.

Some organizations may need to release sensitive documents at specific times (see Section 2.3.4.1). For commercial organizations it may be a press release or new product information that must not be available from the webserver until a specified time. Military organizations may have similar needs to make information available to allies on a timed-release basis. Conversely, a commercial partner or military ally may be an adversary tomorrow in which case they may not be allowed to receive various forms of information.

Other organizations may need to adapt their security policies based on the tasks performed by the users. For example, in the banking example cited above, some tasks may be critical to perform despite the more restrictive policy enforced after 5:00 PM. High-priority or urgent tasks may need to be granted special permissions to complete on-going operations despite the general change of policy. Other task-based policies may make use of an assured pipeline, like that proposed by Boebert and Kain [2]. Assured pipelines address situations in which a series of tasks must be performed in a particular order and the control flow must be restricted. An adaptive policy might change the set of permissions associated with a single process so that, as the process completes one operation, the permission set then allows the process to complete the next operation but prevents it from revisiting objects that it needed to access for earlier operations. A related security policy would be the Chinese Wall introduced by Brewer and Nash [3], which is intended to prevent conflicts of interest in commercial settings. Briefly, under a Chinese Wall security policy a subject may initially be allowed permission to an entire class of objects, but as soon as the subject accesses one element of the class, permissions to access any other object of that class are denied.

[9] The results of the Trade-Off Study were accepted for publication by the 1996 USENIX UNIX Security Symposium in [4].

Another class of examples of adaptive security policies are role-based policies (see Section 2.3.4.2). A role is distinguished from a task in that an individual has an on-going need to complete a set of tasks. In commercial settings, roles may be used to enforce separation of duties (such as purchasing from disbursement of funds). For small companies it may be necessary for one individual to perform actions in more than one role, though not necessarily at one time, to provide proper controls and oversight. Other commercial policies, like Chinese Wall (see [3]), limit the access of a user to certain sets of files to prevent conflicts of interest. In military operations it may be necessary for an individual to perform actions in more than one role simultaneously. In the Navy for example, the role of the Watch Officer on a ship may be performed by the Chief Engineer. It may be necessary for the Chief Engineer to access engineering information as the Watch Officer. Similarly, the Command Duty Officer may need to perform actions reserved for the Commanding Officer in times of emergency. The invocation of such privileges should be restricted for only those times at which they are needed.

A final class of examples in which adaptive security policies is necessary applies primarily to military or intelligence situations which apply multilevel security (MLS) rules. Adaptive policies may allow either a relaxation or selective hardening of confidentiality restrictions (see Section 2.3.4.3). Under MLS rules all objects are labeled according to the sensitivity of the data they contain; e.g. Top Secret, Secret, Confidential, and Unclassified. Users and subjects are allowed access to observe objects only if their clearance level is equal to or exceeds the sensitivity of the object. During an emergency it may be necessary to collapse levels into two levels: Classified for Secret and Top Secret files, and Unclassified for the remainder. Thus, under the relaxed rules someone formerly cleared for Secret could access files formerly labeled as Top Secret. For example, military officers may only have clearance to the level of Secret, but once their troops are under fire, they may need to access Top Secret information such as the location or capabilities of enemy forces. Conversely, confidentiality rules and other security measures could be "hardened up" based on DEFCON alert status or following detection of a possible intrusion. There are a number of ways to "harden up" a system. Among others, one could increase internal controls, perform full audits rather than selective audits, or require additional authentication measures.

Any implementation of adaptive security presents its own set of advantages and disadvantages. Section 5.2 describes the criteria against which implementations of adaptive security may be measured. Section 5.3 describes the range of possible implementations for adaptive security given the basic security architecture of DTOS (background information about DTOS is given in Appendix A and [22]). The final section, Section 5.4, describes in greater detail the four specific implementations researched at Secure Computing Corporation and evaluates each with respect to the criteria from Section 5.2.

## 5.2 Criteria for Evaluation

This section describes the criteria against which the four implementation methods identified above are evaluated.

The goal of providing an adaptive security policy for a computer system is to match the flexibility required by the organization that fields the system. There are two types of flexibility to consider: policy flexibility, the range of policies that a system can support before and after a transition between policies, and functional flexibility, the ability of users to complete tasks despite the transition of policies. However, greater flexibility may come at the expense of security and assurability, and the greater complexity required for some types of transitions may have an impact on the reliability of the system.

The criteria identified here are not independent of one another, in fact examining various implementations for adaptive security leads to a series of trade-offs with respect to these criteria. The conclusions that are drawn from the analysis of the four implementations reflect the nature of the dependence of the criteria upon one another.

Policy Flexibility The DTOS Security Server can enforce a wide variety of security policies [24]. Thus, in one sense the DTOS prototype is "flexible" with regard to the number of security policies that can be enforced. In the context of adaptive security, the concept of policy flexibility could be measured by the amount of change one is allowed to make and whether the system can enforce an arbitrary new policy. Thus, policy flexibility depends on the number of, or lack of, constraints that must be satisfied by the successor policy for a given predecessor policy.

Functional Flexibility Functional flexibility addresses whether the policy transition is graceful or harsh with respect to the applications that are running at the time of the transition. A harsh transition might be like turning off the power and re-booting the system, whereas a graceful transition may appear seemless to the user and most applications on the system. A harsh policy transition may prevent users from performing necessary, possibly urgent, tasks, rather than allowing them to complete their tasks in an evolving security environment. The ideal is to allow necessary tasks to complete while terminating tasks which are not only disallowed under the new policy, but which represent a security risk in the new environment.

Security The existence of a mechanism or method of changing policies may introduce security vulnerabilities. In assessing a method of policy adaptation, this paper will consider the security risks that are inherent in that method of policy adaptation.

Assurability Each type of policy transition will be assessed for the relative difficulty of providing formal assurance evidence in support of the policy transition. To some extent this was discussed in Section 2.6, but this section will add some comments depending on the specific implementation.

Reliability Each method of policy transition introduces a measure of complexity into the system. Changing policy may expose the system to certain risks which decrease the stability of the entire system.

Performance Performance addresses how quickly the policy transition occurs. The ability to change policies quickly has impact on the needs of the user for security, functionality, and reliability. A complex hand-off may allow greater flexibility between policies enforced before and after the transition, but it may also present greater security risks. A less complex hand-off may provide performance gains at the expense of functional grace or of flexibility of specifying security policies.

## 5.3 Implementation Space

The Distributed Trusted Operating System (DTOS) Prototype provides a security architecture that separates the enforcement of the security policy from its definition. Details about the DTOS design are presented in Appendix A and [22]. Since this type of security architecture is not unique to the DTOS Prototype, results from this study will apply to a variety of systems with similar architectures as well.

Elements available to adapt the security policy include the following:

- the number or complexity of the databases that a Security Server uses to initialize its internal state,
- the number of Security Servers available to the microkernel for security computations, and
- the control over which Security Server makes security computations on behalf of the microkernel.

Although the number of possible implementations is large, this study only describes the following representative implementations:

- One Security Server and multiple databases — adapting the policy by forcing the Security Server to re-initialize from a new security database.
- One Security Server and one database — adapting the policy by expanding the internal state of the Security Server and increasing the complexity of the security database to describe more than one set of security policy rules and by providing the Security Server with a mechanism for changing its mode of operation.
- Multiple Security Servers with a single active server providing one point of control over security computations — adapting the policy by providing a mechanism to hand off the responsibility of computing access decisions from one server to another. Thus, one and only one Security Server defines the policy at any given time.
- Multiple, concurrent Security Servers with responsibility for security computations partitioned by tasks — adapting the policy by assigning a pointer to a specific Security Server to each new process. In this method, whenever a process makes a request to the microkernel for service, the microkernel submits requests for access computations to the Security Server which is associated with that process and which defines the security policy with respect to that process.

## 5.4 Comparison of Implementations

This section of the study will describe each of the methods for changing the security policy in greater detail along with the capabilities and limitations presented by each.

### 5.4.1 Loading A New Policy Database

One possible method for implementing a new security policy is to change the way that the Security Server defines it by creating a second database and re-initializing the Security Server. A method for doing this existed on the DTOS prototype already. During the boot process, the microkernel operates on a hard-coded cache of permissions until the Security Server is ready for operation. Once the Security Server has initialized, the microkernel places the command **SSI_load_security_policy** on security port of the Security Server. This command causes the Security Server to read the security database to construct in its internal memory a table that maps SSIs and OSIs to permissions. The Security Server then tells the microkernel to flush its cache of permissions, and from that point onward the policy defined by the Security Server is the policy enforced by the microkernel. The same command can be used to replace one table with another. Once the Security Server has loaded the new policy, it tells the microkernel to flush its cache, and the new policy is enforced by the microkernel.

The command to reload policy can be encapsulated in a user-invoked program or in some automated process which changes the policy at the triggering of some event. Thus, the policy can be changed at regular intervals using a process like the UNIX utility *cron*, or by a background process which monitors the system for intrusion events.

Policy Flexibility This method relies heavily on the tables that can be loaded into the Security Server from the security database. Since the tables are indexed by the SSI and OSI, the management of the system is easiest if the Security Server loads a new policy which is similar to the old one. A radical change of policy requires that each entity in the system have a security context which can be recognized by the active Security Server before and after the policy change.

For initial policies based on Type Enforcement [2] or MLS access rules, it would be difficult to make radical changes in the policy. Every entity that has a type or domain associated with it must also have the attributes necessary for enforcing the different policy. Thus, to change from a policy which enforces a combination of MLS and Type Enforcement rules to a UNIX-like security policy, it would be necessary for objects and processes to have attributes necessary for both sets of security mechanisms. For objects it is necessary to maintain contexts for the type and sensitivity level of the object as well as the users and groups which may have access to the object. For subjects, it is necessary to maintain the domain and clearance level of the subject as well as the user of the subject. It is also necessary to maintain a database listing the group membership.

Functional Flexibility Since the transition between policies during the loading of a new policy is nearly atomic, this implementation is quite harsh on running applications. Any application which ceases to have permission to perform any task under the new rules is essentially orphaned. This abrupt change of behavior is probably acceptable, and may even be desirable in some contexts: military emergencies for example. However, in some contexts this abruptness would cause considerable difficulty. In our banking example for instance, there may be occasions when a particular user must complete a specific transaction before the end of the day. However, if the policy transition time occurs at 5:00 PM sharp and the user needs an additional fifteen to twenty minutes to complete the task, then this implementation of the policy adaptation would hinder bank employees from completing vital tasks. This would be unacceptable under this scenario.

Security Although the security database is a critical object that should be protected from unauthorized modification, a clear security risk is that the security database file could be changed inappropriately while the system is in operational mode. Intuitively, the security database is more susceptible to replacement or modification during operation than the database (and system) would be to attacks conducted between successive boots of the system. If subverted software could replace the intended database with a different file, the system would enforce the wrong policy.

A clear security concern for this type of change of policy is who can authorize, authenticate, and execute the policy change. In the DTOS prototype, authority to reload the security policy is restricted to subjects that have the permission *ss_gen_load_policy*. Authorization to operate subjects with this permission can be restricted to certain processes, to roles, or to sets of individuals with other security mechanisms.

Assurability The immediacy of the transition of this method provides for the greatest assurance: the users always know exactly which policy is the current policy. As will be shown below, this is not always the case with other methods. There could also be some concerns about the flow of information across transitions, but this concern exists for all methods of policy adaptation. Furthermore, some of the formal modeling and proofs might be relatively easier than for more complex transitions.

Reliability A tangible concern is that if the database file has become corrupted, then the Security Server will not be able to read it. The effect of this is that the Security Server dies, and the system is left without any Security Server at all. Not only would the system not be able to enforce the new, intended policy, but the system would have difficulty running at all. The microkernel and other processes that can cache permissions computed by the Security Server would rely solely on the permissions that had been cached up to the time that the Security Server went down.

Both the security and reliability concerns could be ameliorated by placing a checksum (or computing a hash of even a digital signature) over the security database; in this case, the Security Server could be implemented so as not to read in the new database unless the checksum can be verified.

Performance This is the second fastest method for changing policies. During performance testing, a typical transition time (median) required 2.985 seconds, and no transition required more than 3.970 seconds. Although this might not be as fast as necessary in a real-time embedded system, this would be more than satisfactory in systems such as the banking application mentioned in the introduction.

In Figure 5-16, the abscissa ($x$-value) represents the time in seconds required to reload the policy while the ordinate ($y$-value) represents the percentage of observations less than or equal to the $x$-value.

### 5.4.2 Expanding the Database and Security Server State

In this method of transition between policies, when the Security Server loads its initial security database, all of the permissions allowed under all modes of operation would be initialized in the Security Server's internal memory. A mechanism internal to the Security Server would allow it to change policy without having to read a new security database. Thus, policy changes could be triggered by a variety of events. The policy could change based on the time, or when processes complete certain tasks or invoke certain permissions, or when alarms are set off by possible intrusion events. This method is similar to the Reload Policy mechanism above; however, because of the ability to change policies based on triggering events, it has a number of advantages which are listed below.

Policy Flexibility This method has the same restrictions that the Reload Policy mechanism has. It is easiest for the Security Server to alternate among policies which are similar. For initial policies based on Type Enforcement or MLS access rules, the new policy must also be

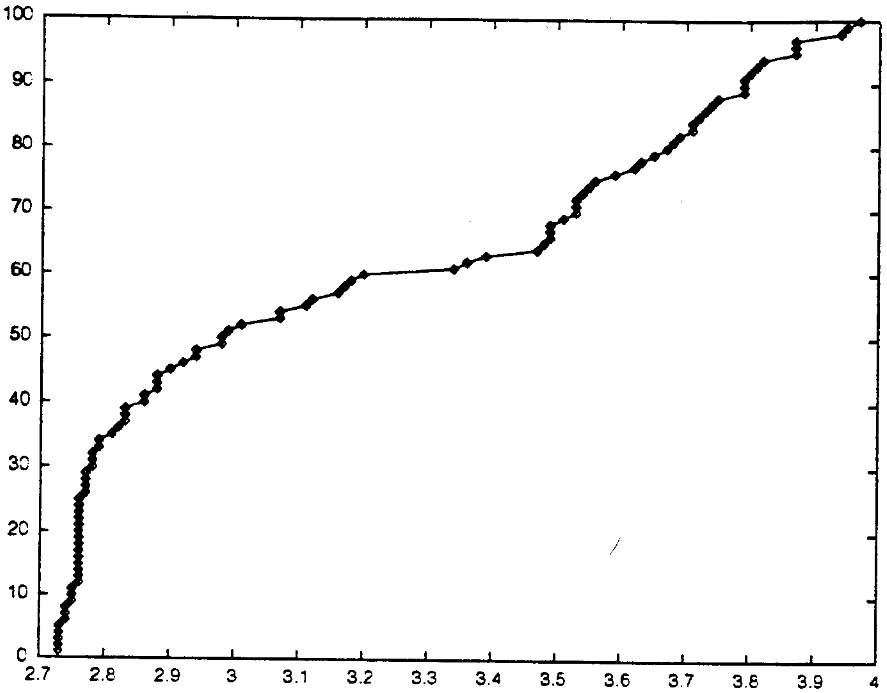


Figure 5-16: The Cumulative Distribution for 100 Trials of the Reload Policy Method

based on Type Enforcement or MLS access rules. However, the mechanisms for changing policy definition give this method greater flexibility than the previous method.

For example, for policies which change on a regular, periodic basis (recall the banking example in which a more stringent policy is enforced for after-hours operation), a timing mechanism that triggers the change of policy could be added to the Security Server.

Another adaptation mechanism could be triggered by the use of particular permissions. For example, when a particular permission is requested and returned to the requesting process, that permission could be removed from the Security Server's notion of the allowed permissions. This would render the permission as a one-time only permission. For example, in a commercial application a one-time permission to issue payment for a purchase order would prevent double payment.

Similarly, when a particular permission is requested and returned to the requesting process, that permission could be removed from the Security Server's notion of the allowed permissions, and one or more could be added. Such adaptations could be chained together. For example, if the Security Server were applying Type Enforcement, a process operating in one domain might be granted access to a new type and denied access to an old one. Thus a set of operations could be performed by a single process in a secure pipeline. Such secure pipelines are already possible with Type Enforcement, but each operation is performed by a separate process, each running in a unique domain (see [2] and [12] for more details). This type of mechanism would also be ideal for enforcing the security policy known as Chinese Wall (see [20] for a description).

Functional Flexibility Since the transition between policies during the loading of a new policy is essentially atomic, this implementation could be as harsh on running applications as the Reload Policy mechanism. However, the database could be expanded to include several policies so that a policy transition could take place with several intermediate policies during the transition. A phased transition of this sort might allow some tasks to complete processing within fixed time limits.

Security  The security concerns here are the same as for the reload policy with the exception that the security database is read once and only once at initialization, and thus the possibility that an untrusted user or process has been able to corrupt it is eliminated.

With the expanded state of the Security Server, changes of policy may be regulated automatically by the time of day, as in the banking example, or by events, as in the Chinese Wall policy [3]. By moving the authority for changing the policy from subjects to events, the methods by which hostile users could alter the enforced policy change. If a hostile user could tamper with the system clock, or force a triggering event, or counterfeit a triggering event, then he could control changes of policy.

The ability to "harden up" system defenses automatically in the event of a possible intrusion also seems to be a particular advantage not present in the Reload Policy mechanism.

Assurability  Again, as with the Reload Policy Method, the immediacy of the transition of this method provides for good assurance, because the users always know exactly which policy is the current policy. However, as with any policy change, there are concerns about the flow of information across transitions. It may be the case, for example, that under a more restrictive policy processes A and B are not allowed to communicate with one another. However, under a more relaxed policy they share access to a common object.

Considering that the policy would change given some triggering event such as the detection of intrusion, it might be possible to capture this type of policy adaptation in the security policy. For example, one might include in the policy the requirement that the policy is hardened in specific ways whenever an intrusion is detected. Thus one could attempt to provide clear arguments about the behavior of the system with respect to these requirements.

Reliability  This method is more reliable than reloading the policy because we are not concerned about the second policy being corrupted after boot-time. However, this method does make the coding of the Security Server more complex which may cause unforeseen problems.

Performance  Explicit performance numbers are not available for this method. However, since it avoids the time-consuming step of reading a new database, it is anticipated to be faster than the Reload Policy Method, and expected transition times should be less than one second. Thus, it is expected to be the fastest of the four methods under discussion.

The microkernel and other processes can cache permissions to improve performance; so changing policy and flushing the cache frequently could cause a minor performance drag. However, permissions in the database can be flagged as noncacheable. Thus, transient permissions as described above could be flagged in that way so that the microkernel would not have to flush its entire cache as it does for the Reload Policy mechanism. Similarly, permissions in the database can be flagged as those which cannot be flushed. Thus, persistent permissions could be flagged so that the microkernel would not have to flush those permissions from its cache at all, and performance would not be adversely affected by the adaptation of policies.

### 5.4.3 Handing Off Control to a New Security Server

In the Security Server hand-off, the current Security Server passes the receive capability for its security port to another Security Server that implements a new policy. In order to accomplish this, the new Security Server is initialized while the current Security Server is still in control of the policy decisions. The new Security Server uses the command **get_special_port** to obtain the send right to client port of the current Security Server and then issues the **transfer_security_ports** to the current server. The current Security Server packages the receive rights for the its security port along with two other tables of information. One contains the mapping between security contexts and SIDs that the current Security Server uses to interpret incoming requests, and the other is a list of the ports of processes which may be caching security permissions. The new Security Server needs the former to interpret requests that it receives regarding any processes or objects that exist prior to the hand-off. It needs the latter because it may eventually need to tell these other processes to flush their cached permissions. The last action of the current Security Server is to tell all processes with cached permissions to flush their caches. At this point the new Security Server can compute access permissions, and the microkernel and any other processes that enforce these permissions can enforce the new security policy.

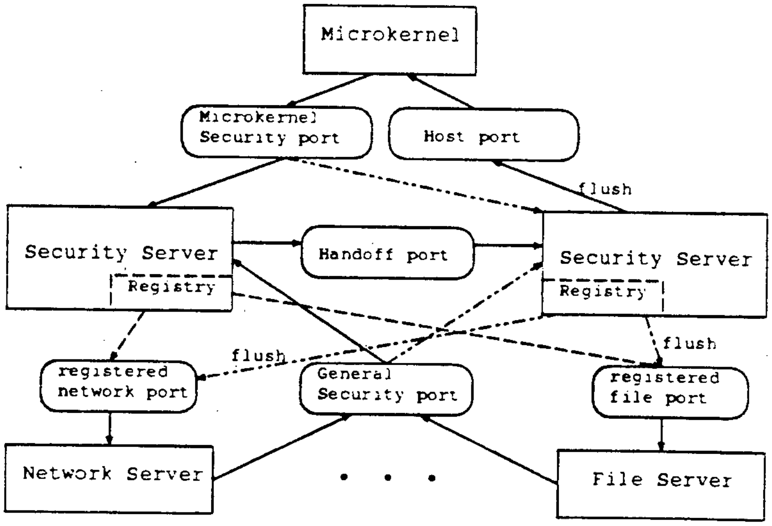


Figure 5-17: Security Server Hand-Off

In order to be able to process new requests for permission computations, the new Security Server must be able to interpret the requests. As mentioned above, the old Security Server sends the appropriate information for the new Security Server to match contexts to SIDs. However, the new Security Server has some knowledge of security contexts prior to receiving this information from the old Security Server; so it must reconcile its understanding of contexts with the mapping information received from the old Security Server. It also must create new SIDs for any new contexts which were not recognized by the old Security Server. For example, if both the new Security Server and old Security Server are implementing Type Enforcement and there are new domains as part of the new policy, each new domain must receive a SID. Similarly, if the hand-off occurs in order to implement dynamic lattices as part of an adaptive MLS policy, any new levels must receive SIDs. Once the new Security Server has completed this reconciliation, the old Security Server can shut down.

Policy Flexibility The greatest strength of the hand-off method is that one can enforce a global, radical change of policy. The new Security Server can implement a very different policy from the one that is enforced before the hand-off. As discussed above in Section 5.4.1, the only impediment to changing the policy in a radical way is the labeling of objects and processes with the appropriate set of attributes which can be interpreted by both the new and old Security Servers. In other words, radically different policies may require essentially disjoint sets of attributes which the system designers glue together for the context of any single entity.

Functional Flexibility In essence this method is not different from the Reload Policy option. Changes to the security policy are global and atomic. The same problems exist in this method as the for Reload Policy for situations where a harsh change of policy is undesirable, as in the banking example.

Security Some of the same security advantages and concerns exist here as for the Reload Policy method. As with the Reload Policy method, the users always know exactly which policy is the current policy. However, if the new Security Server has to initialize from some static file or security database, there is always the risk that it could be subverted. Another possibility is that the code for a new Security Server could be subverted as well and that a malicious Security Server could end up in control of the permission decisions.

Also the question of who can authorize, authenticate, and execute the policy change exists for this method. The Security Server will hand off the security port to the new server when it receives the command *SSI_transfer_security_ports* on its security port. Just as in the case of the authority to reload the policy, the permission to issue this command is restricted to subjects that have the permission *ss_gen_load_policy*. Authorization to operate subjects with this permission can be restricted to certain roles or to sets of individuals with other security mechanisms. The additional concern here is that the security port is transferred to the correct subject, the new Security Server.

Assurability Again, as with the previous two methods, the transition of this method provides for reasonably good assurance. The users know almost exactly which policy is the current policy, but there is a certain lag time while the port rights are in transition. This may lead to the necessity of using temporal logics and arguing about eventuality.

Reliability Unfortunately, the hand-off procedure on the DTOS prototype is delicate, and this is its greatest weakness. The unreliability may be an artifact of the DTOS prototype and the Lites server that is used to provide the light-weight microkernel with services that allow one to use UNIX applications on DTOS. The combination of the microkernel, Lites server, and the Security Server is prone to paging errors and deadlocks. To avoid these errors, the microkernel must have a sufficient set of permissions hard-coded into its cache (these permissions are not flushed from the microkernel). Some of the permissions required by the new Security Server to complete the hand-off must be in the hard-coded cache before the transition is initiated.

For example, the Security Server has pageable memory. During the hand-off, the Security Server may start using new areas of memory while processing a security request from the microkernel. If a page fault occurs, then the Security Server itself will request service from the microkernel. If the microkernel has not cached the permission required by the Security Server, it must in turn request a security computation from the Security Server. However, the Security Server is blocked on the request to the microkernel for service, and the microkernel cannot complete its request without the security computation from the Security Server. What makes these types of events unpredictable is the existence of other processes on the system that may request services from the Lites server while the security port rights are in transit. The new Security Server depends on the Lites server for services, but a thread of execution in the Lites server can be waiting for a security computation creating the deadlock.

Performance This is the slowest of the methods tested. During performance testing, a typical transition time (median) required 4.900 seconds, and all transitions fell with the range of 4.820 to 5.010 seconds. This might not be as fast as the Reload Policy method, but once again this would be more than satisfactory in systems such as the banking application mentioned in the introduction.

In Figure 5-18, as in Figure 5-16, the abscissa represents the time in seconds required to hand off the security port rights to the new Security Server, while the ordinate represents the percentage of observations less than or equal to the corresponding elapsed time.

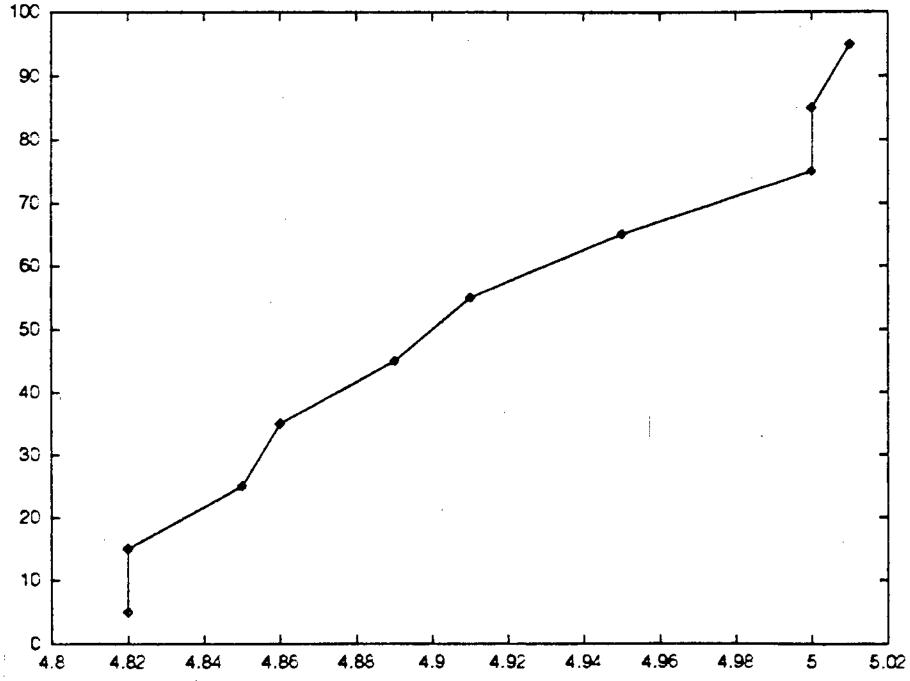


Figure 5-18: The Cumulative Distribution for 10 Trials of the Hand-Off Method

### 5.4.4 Adding Security Servers for New Tasks

The final method for changing the security policy is to create a set of task-based Security Servers. With this method there may be more than one Security Server computing access decisions for the microkernel and other clients, each defining a separate set of security rules. While the microkernel is enforcing multiple policies, each task on the system is associated with one and only one Security Server, and therefore, each task operates under a single policy.[10] In

[10] It is not exactly true that each task has only one Security Server, but it is a useful fiction for the time being The bottom line is that there is only one way for each access request to be computed by the entire set of Security Servers.

the three previously described methods, all tasks operate under a single, monolithic policy.

For this method we introduce a new global variable: the Security Server Stack[11] Each entry in the stack consists of a data structure containing the security and client ports for each Security Server. At boot time, the initial Security Server uses the **set_special_port** command to enter the security and client ports to the stack in the 0-th place. Another global variable, **curr_ss**, points to the 0-th entry in the stack to indicate that the initial Security Server is the *current* Security Server. When another Security Server is created, it also enters its ports to the stack at the first available entry, and **curr_ss** is incremented to the next position in the stack.

Each task has a pointer labeled **ss_ptr** that identifies the Security Server that defines the policy under which the task is running. When tasks are created, **ss_ptr** is set to **curr_ss** by default, though the parent task may cause the value of **ss_ptr** for the new task to be set to the parent's Security Server. Like any other process, each new Security Server itself operates under the policy defined by a Security Server which precedes it in the stack (the Security Server immediately preceding it would be the default). When the microkernel receives a request, it checks its cache for the permission. Permissions in the cache are identified by a triple: two SIDs, as before, and the ss_ptr of the requesting subject. If the permission is not in the cache, it sends a request to the Security Server assigned to the requesting task. The Security Server computes the requested security access, unless it receives a request with a context that it does not understand. If the Security Server cannot resolve the SIDs into security contexts, it forwards the request to its own Security Server.[12] The request is passed down the stack until some Security Server is able to resolve the SIDs into contexts and a security computation can be made.

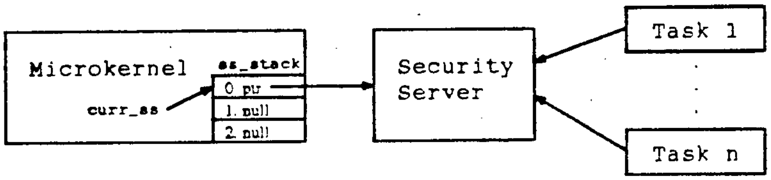


Figure 5-19: Security Server Stack Before "Push"

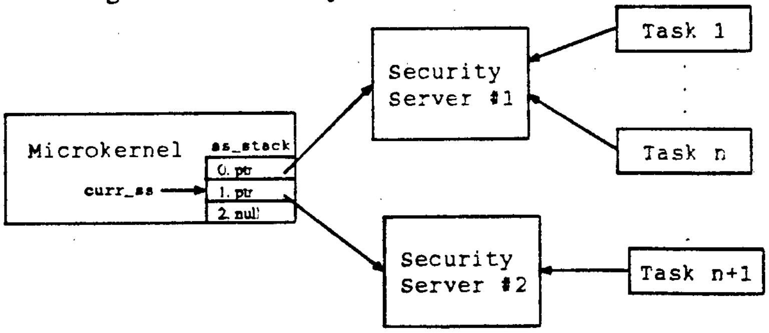


Figure 5-20: Security Server Stack After "Push"

This method for changing the security policy is the most robust and possibly the most flexible method of the four methods discussed in this study. However, the additional flexibility and

[11] Like other processes, each Security Server refers to a preceding Security Server. If each Security Server in the stack refers to its immediate predecessor in the stack, then it is truly a stack-like implementation. If the Security Servers in the "stack" refer to servers older than their immediate predecessors, then a graph of the dependencies could be more accurately described as a "tree."

[12] This is the reason that tasks do not necessarily have only one Security Server.

reliability of enforcing multiple security policies may come with an increased cost for assuring the security of the system.

Policy Flexibility This method for changing policy provides the capability for considerable flexibility for changing the policy. However, as new Security Servers are created, only new tasks operate under the new policy rules; so changes to the system-wide policy are local rather than global. In other words: you can't teach an old dog new tricks, because old tasks will continue to run under the policy defined by the old Security Server.

There is the possibility that the stack could be augmented by using one of the other policy changing mechanisms to force old tasks to run under a new policy. For example, if there are two servers in the stack at positions 0 and 1, the Security Server at position 0 could hand off to a third Security Server which is identical to the Security Server in position 1. Thus, both servers operating would define the same policy, and the microkernel would be enforcing only one policy rather than two. (In fact, the first two servers could then exit, all tasks with pointers to the second Security Server would be re-directed to the server at position 0 (the third server), and the system would only have one Security Server as well as one policy.)

Functional Flexibility Functional flexibility is the greatest strength of the Security Server stack method. Allowing running processes to run under their original policy is a way of "grandfathering" in their allowed accesses. Thus, in our banking example, if some user is actively working on a task at 5:00 p.m. which must be completed, but the bank's security policy is set to change to a more restrictive policy at that time, the user would be allowed to continue his task because the task is operating under the less restrictive policy. However, any attempt by a user to create new tasks after 5:00 pm would be subject to the new, more restrictive policy.

Security This method is a double-edged sword. It is possible that certain tasks which need to be highly constrained could operate under more restrictive policies than is generally allowed. This could be an advantageous design for increasing security. However, once a task is granted a permission to perform some operation it is allowed to keep it, even if another, more restrictive Security Server is pushed onto the stack. Thus, in the event of an intrusion, a rogue process which has gained unauthorized access to system resources may be able to continue unchecked. Thus, the gains made for functional flexibility allow for a loss of security. In order to harden up the defenses of a system like this, it would be necessary to graft another method of policy change on top of this one.

Assurability Coordinating the necessary elements to implement this method could be a nightmare for system designers and for any attempts to provide formal assurance evidence. Furthermore, there would be multiple, overlapping security policies. One could not make broad global statements about the behavior of the system and the rules in place at any given time; however, one may make statements on a per task basis. For this method, it may be acceptable to do this, and it would even reduce the necessity of having to use temporal logics and having to make difficult arguments about eventuality.

Reliability This method improves upon the Hand-off Method for reliability because there is no vulnerable moment when the rights for the security port are in transit. It is also more reliable than the Reload Policy Method because the top Security Server in the stack will still be able to make security computations even if new Security Servers fail to initialize due to corrupted security databases.

Performance Explicit performance numbers are not available for this method. However, it is anticipated to be as fast or faster than the Hand-off Method, and expected transition times should be between four and five seconds. The greatest factor in the performance for the Stack Method is the loading of the large executable for creating a new server to push onto the stack, but this is also true of the Hand-off Method. The Hand-off Method is slower because the rights to the security port have to be transferred from one server to the other. This is quicker than loading the executable for the new server, but adds an extra wait.

## 5.5 Conclusions

For security architectures which separate the definition of the policy from its enforcement, the solution space for implementing adaptive security policies is large. From the entire range of such implementations, this study has examined four possible methods which have been implemented, or partially implemented, for the DTOS prototype by Secure Computing. Each implementation has strengths and weaknesses. The criteria for evaluating these methods are described in greater detail above, but the trade-offs are encapsulated in Table 5-7 below. From the table the Stack Method and the Expanded State Method appear to be the most attractive options for implementing adaptive security, but which choices one makes depends on the eventual application for the implementation as suggested below.

| | **Implementations** | | | |
|---|---|---|---|---|
| **Criteria** | Reload Policy | **Extended State** | **Hand-Off** | Server Stack |
| Policy Flexibility | fair | good | fair | excellent |
| Functional Flexibility | poor | good | fair | excellent |
| Security | good | excellent | fair | poor |
| Assurability | excellent | good | fair | poor |
| Reliability | fair | excellent | poor | good |
| Performance | good | excellent | poor | fair |

Table 5-7: Summary of Trade-Offs

When applied appropriately, the Reload Policy and Expanded State methods are the lightest weight implementations and provide good features for a narrow subset of applications. In particular, the key features of these two methods are that they allow the Security Server to reload a database, but they do not alter the algorithms by which the Security Server makes its security computations. The database and Security Server implementation for the Expanded State method has the potential for becoming complex. The additional complexity posed by this work may make alternate methods for implementation more attractive. The Expanded State method is best left to small, incremental changes to the policy. By comparison the Reload Policy Method is probably not an attractive option for systems in which there are large numbers of small changes to the policy databases since each change of policy would require its own database, and the issue of scalability may be burdensome.

The other two methods, the Hand-Off and the Stack Methods, allow for changes to the algorithms for computing permissions, and this is what accounts for a greater degree of policy flexibility. Because of the multiple points of control, the Stack Method offers the greatest functional and policy flexibility, and the inheritance structure of the parent-child relationships between Security Servers offers the ability to grandfather permissions for running applications. However, that very same asset is a liability. Policy changes under the Stack Method are local, not global. Thus, it is not possible to revoke permissions using that method alone. Furthermore, depending on the number of policies supported on the system, the Stack Method holds the potential for being the heaviest weight implementation.

Not addressed in the discussion of individual implementations, nor in Table 5-7, is the possibility of mixing and matching the four methods to capture the best security features of one method with the best flexibility features of another. For example, one might combine the Stack and Hand-Off Methods in the following way. Tasks would operate under task-based policies with the Stack Method up to a certain point in time, allowing for local changes to the policy based on roles and tasks, and then a server might hand off to its parent and shut down. For example, in the banking application in which the more restrictive nighttime policy is the child of the less restrictive daytime policy (i.e., the stricter Security Server is pushed onto the stack at 5 PM), the nighttime server could hand off to its parent the following morning at 8 AM and shut down. Similarly one might follow the Hand-Off or Stack Methods with a Reload Policy to change the internal tables of a Security Server without changing the fundamental algorithms by which it operates.

Section 6

# Summary

## 6.1 Conclusions

Each major section of this paper is accompanied by a subsection of conclusions. The following is a thumbnail sketch of more detailed conclusions which can be found there.

### Tranquility Study

The tranquility study listed scenarios requiring adaptive scenarios and common tranquility assumptions made about non-adaptive systems. The study discussed specific scenarios requiring adaptive security and the tranquility assumptions that such scenarios would violate. One case of particular interest that was investigated was dynamic lattices, where the level POSet is not tranquil. The study proposes two methods for representing a security lattice which would support inclusion of extra levels and a policy for recovering from a period of policy relaxation.

One method for representing the security lattice is the usual method of combining *levels* and *categories*. In this method a number of artificial categories are added without changing the number of recognized labels (points in the lattice) and without changing the partial ordering. As many artificial categories may be added as there are existing lattice points so that new points can be added to the lattice. New security labels are constructed through the manipulation of the expanded set of categories.

The second method for representing the security lattice uses the local dominance structure. Since a security lattice is a directed graph, the relative location of each point in the lattice is described by listing all of the other points in the lattice that are directly above it and directly below it. It suffices to list only those points directly above or those points directly below; so there is some redundance in the description. However, new points may be added to the lattice simply by listing the points in the original lattice between which the new points will exist.

The high-water mark confidentiality audit policy corresponds to a type of integrity policy in the way that the Bell and LaPadula confidentiality policy corresponds to the Biba integrity model. It adds a contamination label, which is drawn from the same set of labels as the security labels, to subjects and objects. During a period of policy relaxation, as a subject or an object is exposed to entities with a higher contamination label, its contamination label changes to match; therefore the contamination label increases monotonically to record the possible level of contamination for that subject or object. At the time of recovery, subjects or objects whose contamination label is different from their security label can be audited to verify the actual contamination of that entity.

The tranquility study also examined a set of typical tasks performed to provide formal assurance evidence and considered how these specific tasks would be affected by the loss of tranquility. The tasks considered include policy modeling, formal specification, proofs of security requirements based on formal specification, spec-to-code analysis, and covert channel analysis. A conclusion which crosses all of the boundaries of this analysis is that it may be necessary to formalize the security policy or write specifications using temporal logic. The use of such logics makes it difficult to write both the formal model of the security policy and the formal specification of the system. Consequently, reasoning about the requirements from the specifications is also more difficult. Policies using temporal or real-time logics would be likely to be less comprehensible in global terms. Specifications may never be able to accurately model some behaviors. Proofs that argue about eventuality and fairness are difficult. Difficult specifications make spec-to-code more difficult. Finally, the current state-of-the-art in covert channel analysis does not encompass adaptive security: there is no theory for conducting such an analysis, and existing tool support assumes a static policy. However, one conclusion from this portion of the study is that, while some generalizations can be stated about how the formal assurance might need to change, some of these tasks need to be performed for concrete system in order to fully understand the full impact that adaptive security has on assurance.

### Audit

The investigation of audit techniques produced two promising results. Each of these results arises from particular problems faced during the collection and interpretation of audit information. The first problem is that by auditing fine-grained permission checks, it is difficult to relate these to a single chain of execution. The second problem is that permission checks are at too low a level to be related to functional operations.

A solution for the first problem was the introduction of a tracing identifier (TID). The TID is added to the microkernel thread structure, and thus it accompanies a series of requests through the machine as threads pass messages requesting service of other entities. The TID is included in the audited information, thus allowing the audit manager to organize audited events into sets from which higher-level actions can be inferred and analyzed.

A solution for the second problem is to alter the microkernel to monitor service requests made of other servers. Thus, the microkernel may be described as "snooping" the messages sent to those servers to look for auditable events.

One other result of the work on auditing is that audit events may be used as triggers for policy adaptation. Some minor changes were made to the Security Server and the Audit Daemon so that when certain events are audited, the Audit Daemon can inform the Security Server which then changes the definition of the security policy. This could be useful for hardening a policy following intrusion detection or for implementing a commercial security policy like Chinese Wall.

### Database Tools

Maintaining security database files for any complex system requires the use of some management tools. A typical method for creating and maintaining security databases is to write specifications for the database in a specialized language and to use tools to compile the specifications into the actual database that the system uses for making access computations. The existing procedure for DTOS has been to write several files specifying the security policy and then to run the files through the M4 macro processor and some Perl scripts to generate access vectors. One problem with this method is that the macros used to logically group permissions do not afford simple modifications. Furthermore, both the input and output files use syntaxes which are difficult to read and interpret.

For this task, a tool with a graphical user interface (GUI) for specifying the security database was designed and implemented. The advantages of such a tool are two-fold. First, the tool supports the maintenance and modification of security database information which is a costly and error-prone operation when performed "by hand." Second, by using a tool to specify the security database directly, the user need not learn a specification language. The GUI tool provides the means for creating and modifying the policy along with the compilation tools needed to generate access vectors.

Four primary requirements were identified for the tool.

1. The tool must permit grouping of permissions into hierarchical sets.
2. The tool must support specific adaptation methods.
3. The tool must allow for upgrading existing DTOS policy files.
4. The tool must minimize modification to the Security Server by using a format for output files which are compatible with existing files.

The first requirement is a general requirement which might be imposed for any specification tool. The second requirement is specifically related to the nature of adaptive security. The final two requirements were imposed because of existing constraints from the development environment.

The tool was developed using the Windows 95/Windows NT system because of the wealth of tools available for this platform. The tool developed supported a policy adaptation method using time-of-day adaptations.

### Trade-Off Study

Four possible implementations of adaptive security were compared relative to six criteria. The implementations considered included reloading the security policy of the Security Server, expanding the state of Security Server to include more than one policy, forcing the security Server to hand off control to another Security Server, and implementing concurrent Security Servers each defining the security policy for a subset of the running processes. The criteria include the flexibility to change policy, the effects on running processes, security, assurability, reliability, and performance.

The trade-off study does not recommend a single solution for adaptive security policies; instead the results indicate that developers have a range of choices which allow them to pick one or more solutions which best fit their needs. In particular, a system with a single Security Server, can be used for simple policy transitions. This works especially well for policy transitions that must occur globally and quickly. Such policy adaptations may also provide greater security in the face of intrusion detection. More complicated implementations are needed for other scenarios. For example multiple, task-based security servers can be employed with the benefit that policy changes are local in a way that would support grandfathering.

## 6.2 Lessons Learned

The main lesson learned from exploring the impact on assurance from the loss of tranquility assumptions was that a number of formal assurance tasks have to be performed on an actual system with an adaptive policy in order to fully understand the topic. Concrete results will only be gained from working on a concrete system.

The DTOS Medical Demo was analyzed to get a more concrete understanding of the impact of the loss of tranquility. However, since the assurance tasks for this example system were not actually carried out, the analysis was still quite general.

In assessing the usefulness of audit data, it was learned that auditing the low-level permission checks in the microkernel are not sufficient to trace the flow of information. While auditing fine-grained permission checks in the entire system would be sufficient, it would also be costly. Since the Lites UNIX server has some security features, auditing permission checks by the microkernel and the Lites server would be sufficient for a larger set of operations though it is clearly not sufficient for some cases. In particular, Oracle$^{TM}$, a database management application, enforces its own permission checking and does not rely upon the file system.

Two of the requirements stated for developing the database tools are somewhat incompatible. Namely, the goals of specifying the policy at a level which is high enough to be understandable and yet to maintain control of permissions at a low level.

In the trade-off study, it was learned that performance was a relatively minor issue when considering the criteria on which to base choosing an implementation of adaptive security. The policy and functional flexibility of the various methods were more important issues. The appropriateness of the implementation depended almost wholly on the scope and types of the change required and the needs of the organization.

## 6.3 Future Work

Current work on adaptive security has focused on theoretical aspects of adaptive security policies and on various mechanisms for implementing adaptive security. Future work on adaptive security policies should turn from the theoretical to the applied, hopefully by implementing a demonstration system. For example, one might implement a set of banking applications that would operate under policies for daytime and after-hours processing. A demonstration system of this type should also be accompanied by formal assurance evidence such as a formal security policy. However, until there is a real system to examine, formal assurance for adaptive security can only be speculative.

For audit, the use of the tracing identifiers (TIDs) appears to be a good suggestion, but they have not been tested in a system in which the administrator is attempting to recover from relaxed security. However, there are other proposals for future work beyond giving greater meaning and form to the discussion of assurance and audit.

The the Stack Method presented in the trade-off study appears to be very interesting. Since role-based access control (RBAC) is a topic of current interest in the computer security community, further study of the ability of the Stack Method to support RBAC would be beneficial. The inheritance-like relationship between parent-child pairs of servers would be of particular interest.

Other areas of future research could include

1. automating tools for recovery from policy relaxation
2. build a database specification tool with event-based policy adaptation mechanisms
3. implement more fully two of methods described in the Trade-off Study: expanding the Security Server state and the Security Server Stack

# *Appendix* A DTOS Overview

DTOS was designed around a security architecture that separates enforcement from the definition of the policy that is enforced. This architecture allows the system security policy to be changed without altering the enforcement mechanisms. The policy is defined as a function from the security context of the subject making an access and the security context of the object being accessed to a set of permissions. Currently, DTOS implements security contexts consisting of level, domain, user, and group, but the set of attributes that form a security context is configurable. Enforcement consists of determining whether the permissions specified by the policy are adequate for an access being attempted. The generality of the DTOS security architecture has been studied as part of the DTOS program [24]. The conclusion of this study is that a large variety of security policies, useful for both military and commercial systems, can be implemented.

The basic DTOS design is a microkernel, which implements several primitive object types and provides InterProcess Communication (IPC), and a collection of servers which provide various operating system services such as files, authentication, and a user interface [8, 16]. Of particular interest is a *Security Server* that defines the policy enforced by the microkernel and also possibly by other servers. When a request is made for a service provided by the microkernel, the microkernel sends identifiers for the security contexts of the subject and of the object to the Security Server. These identifiers are referred to as *security identifiers* or *SID's*. A context contains attributes about a subject or object that are necessary for making security decisions. For example, the context may contain the domain of a subject or the type of an object, or the level of a subject or object. The information that makes up the context is dependent on the policy; the actual contexts are local to the Security Server and are not available to the microkernel. The Security Server then computes permissions for the context pair, as defined by the policy that it represents, and replies to the microkernel. The microkernel is ignorant of the context of each entity since it only enforces the permissions that the Security Server computes on its behalf. Finally, the microkernel determines if the permissions required for the request were present in the reply. Other servers can communicate with the Security Server in a similar fashion.

For example, a Security Server implementing an MLS policy might maintain subject and object contexts consisting of a level. For the microkernel to enforce the simple security and $*$-property of the Bell and LaPadula model of confidentiality, the Security Server would only grant a write permission if the level for the object security identifier dominates that of the level for the subject security identifier and read permission if the level for the subject identifier dominates that for the object identifier (both permissions are granted if the levels are equal). A file server would check for write permission before allowing a request to alter a file. Alternatively, a Unix-style Security Server might maintain a user and a group for each subject context and an owner, group, and access control bits for each object context and grant permissions from the access control bits depending on whether the user in the subject context matches that of the owner and whether the groups match.

A prototype DTOS microkernel and Security Server has been built by Secure Computing. The microkernel is based on Mach, developed at Carnegie Mellon University [14, 21]. A version of the Lites UNIX emulator, modified by the Government, provides secured UNIX functionality.

The object types implemented by the microkernel include *task*, *thread*, and *port*. Tasks and threads represent the active subjects, or processes, in the system. Each task has a security context that is used for security decisions involving that task. The state of each task includes a virtual memory consisting of a set of disjoint memory regions, each of which is backed by a server that is used to swap pages of the region in and out of physical memory. Each task contains a collection of threads, each of which is a sequential execution, that share the task's virtual memory and other resources. A server is implemented as one or more tasks.

The ports are unidirectional communication channels that the tasks use to pass messages. Tasks use *capabilities* to name ports, and these are kept in an IPC name space on a per task basis. Each capability specifies the right to either receive from or send to a particular port. These capabilities may be transferred to another task by sending a message. For each port, there is exactly one receive capability and therefore at most one task can receive from the port (no task is able to receive from a port for which the receive capability is in transit rather than in an IPC name space). IPC is asynchronous in that messages are queued in the port and the sending task does not wait until its message has been received (an exception is when the microkernel is the receiving task, in which case the sender waits until the microkernel finishes processing the message).

Sending or receiving a message is a Mach microkernel operation to which DTOS has added security controls that enforce the security policy. Thus, possession of the appropriate capability for a port is necessary but not sufficient in order to send or receive a message from that port. The security contexts of the task and the port must also permit the operation. The policy also constrains what capabilities may be passed in a message sent or received by a task.

The Security Server receives requests from the microkernel through the *microkernel security port* and from other servers through a *general security port*. Requests contain an operation identifier (allowing the Security Server to record which permissions have been requested in support of history-based policies that depend on the sequence of operations made on an object), a subject security identifier SSI (representing the security context of the subject), an object security identifier OSI (representing the security context of the object), and a send capability for a reply port. The Security Server replies by sending the permissions for that pair to the reply port (Figure A-21). Not shown in this figure is the fact that the Security Server both defines and enforces a policy for the requests that it receives. It might allow security determination requests from some subjects, but not from others. Similarly, it might allow security determination requests from a particular subject only for certain (SSI,OSI) pairs.

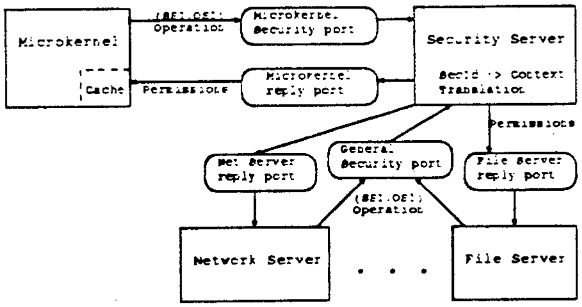


Figure A-21: Security Server Interaction

Security enforcement as described above would be very expensive due to the large number of messages that must be exchanged between the microkernel and the Security Server. The solution in DTOS is to cache (SSI,OSI) pairs with their permissions in the microkernel [16].

When the microkernel receives a request, it first looks in the cache for the appropriate (SSI,OSI) pair. If that pair is in the cache, the microkernel uses the cached entries. Otherwise, it sends the pair to the Security Server to determine the permissions, usually also caching the reply (part of the permission set returned is permission to cache the reply — caching would not be permitted for permissions granted for a single operation by a dynamic policy). Since sending to and receiving from a port are microkernel operations controlled by the policy, the cache must be preloaded with permission for the Security Server to send and receive from the designated ports.

In order to implement a different policy, either by changing the current Security Server or by referring to a new Security Server, there must be a mechanism for flushing permissions from the microkernel's cache. Otherwise, if the new policy removes permissions from the system for a specific (SSI, OSI) pair, and the microkernel has already cached the permissions for that pair, then the microkernel would continue to enforce the old policy rather than consult the Security Server defining the new policy. Therefore, the Security Server may issue a command to the microkernel, and any other servers registered as caching permissions determined by the Security Server, telling it to flush its cache. However, it would be impractical for the microkernel to flush every permission in its cache; for if it did, the entire system would come to a halt. Therefore, some permissions are hard-coded. These include some of the basic permission required for IPC between the subjects comprising the operating system itself.

The separation between policy and enforcement in the DTOS prototype made it attractive for studying adaptive security. The work described in this report discusses refinements to the design that are important for these policies.

# Bibliography

[1] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings 8th National Computer Security Conference*, pages 18–27, Gaithersburg, MD, October 1985.

[2] W.E. Boebert and R.Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, September 1985.

[3] David F. C. Brewer and Michael J. Nash. The Chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA, May 1989.

[4] Michael Carney and Brian Loe. A comparison of methods for implementing adaptive security policies. In *7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[5] National Computer Security Center. Department of Defense Trusted Computer System Evaluation Criteria. Technical report, US National Computer Security Center, Fort George G. Meade, Maryland 20755-6000, December 1985.

[6] National Computer Security Center. Integrity in Automated Information Systems. Technical Report 79-91, US National Computer Security Center, Fort George G. Meade, Maryland 20755-6000, September 1991.

[7] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, CA, April 1987.

[8] Todd Fine and Spencer E. Minear. Assuring Distributed Trusted Mach. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 206–218, May 1993.

[9] Simon N. Foley. The Specification and Implementation of 'Commercial' Security Requirements Including Dynamic Sepregation of Duties. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, Zurich, Switzerland, 1997.

[10] Simon N. Foley, Li Gong, and Xiaolei Qian. A Security Model of Dynamic Labeling Providing a Tiered Approach to Verification. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1996.

[11] J. Gougen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1982.

[12] Paula Greve, John Hoffman, and Richard Smith. Using Type Enforcement to Assure a Configurable Guard. In *Proceedings of the 13th Annual Computer Security Applications Conference*, 1997. To appear.

[13] Geoffrey R. Hird, Daryl McCullough, Stephen Brackin, and Doug Long. Research advances in handling adaptive security. Technical Report RL-TR-95-92, Rome Laboratory, June 1995.

[14] Keith Loepere. *OSF Mach Kernel Principles*. Open Software Foundation and Carnegie Mellon University, May 1993.

[15] Terry Mayfield. Electronic mail addressed to Cornelia Murphy <murphy@securecomputing.com>, December 1996. Re: Security policies.

[16] Spencer E. Minear. Providing policy control over object operations in a Mach based system. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 141–156, June 1995.

[17] NCSC. Trusted computer systems evaluation criteria. Standard, DOD 5200.28-STD, US National Computer Security Center, Fort George G. Meade, Maryland 20755-6000, December 1985.

[18] Richard C. O'Brien and Clyde Rogers. Developing applications on LOCK. In *Proceedings 14th National Computer Security Conference*, pages 147–156, Washington, DC, October 1991.

[19] Owre, Shankar and Rushby. The PVS Specification Language (Beta Release). User Manual, SRI International Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, June 1993. http://www.csl.sri.com/reports/pvs-language.dvi,ps.Z.

[20] Charles P. Plfeeger. *Security in Computing*. Prentice Hall, Inc., Upper Saddle River, NJ, 2 edition, 1997.

[21] Richard F. Rashid. Mach: A case study in technology transfer. In Richard F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, chapter 17, pages 411–421. ACM Press, 1991.

[22] Edward A. Schneider, William Kalsow, Lynn TeWinkel, and Michael Carney. Experimentation with adaptive security policies. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, January 1996. Final Report for Rome Laboratory contract F30602-95-C-0047.

[23] Secure Computing Corporation. DTOS Demonstration Software Design Document. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, July 1995.

[24] Secure Computing Corporation. DTOS Generalized Security Policy Specification. DTOS CDRL A019, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1997.

[25] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 1992.

[26] D. Sutherland. A model of information. In *Proceedings 9th National Computer Security Conference*, 1986.

☆U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-61194